Pattern Recognition in Procedural Knowledge

Breandan Considine

January 26, 2021

Contents

1 Introduction **2**

2 From exact to approximate equality **3**
2.1 Decidability 4
2.2 Intensional equivalence 5
2.3 Computational equivalence 6
2.4 Observational equivalence 7
2.5 Approximate equivalence 7
2.6 Probabilistic equivalence 9

3 From computation to graphs **12**
3.1 Programs are graphs 12
3.2 Distributions are graphs 13
3.3 Knowledge is a graph 14
3.4 Algebraic graphs 15

4 From code to procedural knowledge **16**
4.1 Documentation alignment 17
4.2 Code search 18
4.3 Program synthesis 19
4.4 DSL generation 19

5 Acknowledgements **20**

1 Introduction

Historically, most knowledge was stored as natural language. A growing portion is now *code* [1]. The majority of code is procedural knowledge, written by a human and intended to operate a machine. Though it shares many statistical properties in common with natural language [23], code has an unambiguous grammar and well-defined semantics [41]. We can use these properties to precisely reason about operational or procedural correctness.

Prior work explored differentiable programming [10]. 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*. We propose a set of tools and techniques for reasoning about uncertainty in the form of *procedural knowledge*. In our work, we define procedural knowledge as a graph whose topology represents how to structure computation in order to achieve some desired programming task.

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 [14], and simulate or execute them on various input data [8] 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 similar 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 [40], both enabling a large class of useful data transformations and vastly simplifying their implementation on physical machinery. Depending on the author's perspective, it is possible to view the evaluation of these routines as either operating a stateful machine, reducing a symbolic expression or running a dynamical process on a dataflow graph.

In this work, we first define exact and approximate equality and cover some deterministic and probabilistic algorithms for deciding it (§ 2). We then describe a few graph representations for encoding approximate procedural knowledge (§ 3). Finally, we will discuss some opportunities for applying these ideas to search-based software engineering, in particular, documentation alignment, code search, synthesis and DSL generation (§ 4).

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 is equality. Equality is like identity for objects that may differ in superficial ways, but are the same in all but name and appearance.

For such a fundamental concept, the notation for equality is recklessly overloaded in mathematics and computer science. Depending on context, $x = y$ 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 should be 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, $\delta_k^{-1}(B \subset \mathbb{B})$ is not open, thus δ_k is not continuous and $\nabla\delta_k$ is undefined. Now consider the Dirac δ-function, which is defined as follows:

$$\forall f \in \mathbb{R}^2 \to \mathbb{R}, \int_{\mathbb{R}^2} f(x, y)\delta_d(x - a, y - b)d(x, y) \triangleq f(a, b)$$

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 δ_k nor $\nabla\delta_d$ are a satisfactory basis for logical equality. To allow a more flexible definition of the $\stackrel{?}{=}$ 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* $\equiv$, a binary relation with the following logical properties:

$$\frac{}{a \equiv a} \qquad \frac{a \equiv b}{b \equiv a} \qquad \frac{a \equiv b \qquad b \equiv c}{a \equiv c} \qquad \frac{a \equiv b}{f(a) \equiv f(b)}$$

Identity *Symmetry* *Transitivity* *Congruence*

2.1 Decidability

To determine if two procedures are equal, we need a decision procedure. We list several high-level approaches for deciding exact and approximate equality in the deterministic and probabilistic setting in the table below:

	Deterministic	**Probabilistic**
Exact	Type Checking	Variable Elimination
	Model Checking	Probabilistic Circuits
Approximate	Software Testing	Monte Carlo Methods
	Dynamic Analysis	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 [17] and higher order theories [18]. We know there can be no machine which accepts every equality and rejects every disequality in a universal language [49]. By extension, any nontrivial property of partial functions is undecidable [43].

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 $n \in \mathbb{N}$.

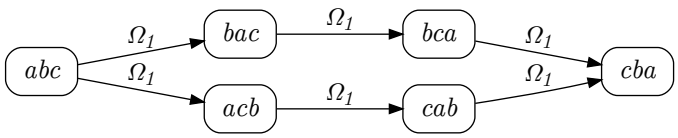
$$\frac{f, g : \mathcal{X} \to \mathcal{Y} \in \Gamma_g^0}{\Gamma_g^0 \vdash \{f\}, \{(f, f)\}}\ \textsc{Init} \qquad \frac{\Gamma_g^n \vdash E \subseteq \mathcal{F}, G \subseteq E \times E}{\Gamma_g^{n+1} \vdash \bigcup_{\substack{e \in E \\ \sigma \in \Omega}} e' \leftarrow e[\sigma_1 \to \sigma_2],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

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

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:

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

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.

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

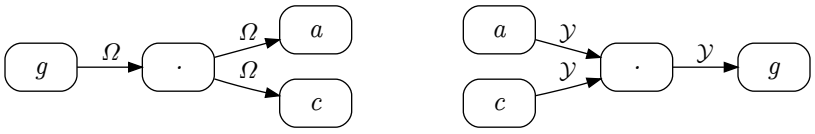
$$\frac{\Gamma, \Gamma' \vdash f(\Omega) \Downarrow g(\Omega)\ \forall \Omega}{\Gamma, \Gamma' \vdash f \equiv g}\ \textsc{Eq} \qquad \frac{\Gamma, \Gamma' \vdash \exists \Omega \mid f(\Omega) \not\Downarrow g(\Omega)}{\Gamma, \Gamma' \vdash f \not\equiv g \text{ by } \Omega}\ \textsc{Neq}$$

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 Inv. Let us consider $f : \{a, b, c\} \mapsto abc, g : \{a, b, c\} \mapsto ac$ under $\Omega := \{(a, 1), (b, 2), (c, 2)\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f\ := abc, \Omega := \{(a,1),(b,2),(c,2)\} \in \Gamma}{\Gamma\ \vdash\ a(\Omega) \cdot bc(\Omega)}\ \textsc{Inv}}{\Gamma\ \vdash\ 1 \cdot bc(\Omega)}\ \textsc{Sub}}{\Gamma\ \vdash\ 1 \cdot b(\Omega) \cdot c(\Omega)}\ \textsc{Inv}}{\Gamma\ \vdash\ 2 \cdot c(\Omega)}\ \textsc{Sub}}{\Gamma\ \vdash\ f(\Omega) \Downarrow 4}\ \textsc{Sub} \qquad \dfrac{\dfrac{\dfrac{g\ := ac, \Omega := \{(a,1),(b,2),(c,2)\} \in \Gamma'}{\Gamma'\ \vdash\ a(\Omega) \cdot c(\Omega)}\ \textsc{Inv}}{\Gamma'\ \vdash\ 1 \cdot c(\Omega)}\ \textsc{Sub}}{\Gamma'\ \vdash\ g(\Omega) \Downarrow 2}\ \textsc{Sub}$$

$$\Gamma, \Gamma' \vdash f \not\equiv g \text{ by } \Omega := \{(a,1),(b,2),(c,2)\}\ \textsc{Neq}$$

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



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

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

Although a single witness Ω s.t. $f(\Omega) \not\Downarrow g(\Omega)$ is sufficient for disequality, this procedure may be intractable depending on $|\mathcal{X}|, P_{\text{NEQ}}(\Omega)$ and Θ_{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* [37] 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

Though precise, boolean equality is far too rigid for many applications. The spaces involved either lack formal semantics or are intractable to verify, leaving decidability out of reach. We can relax this definition by introducing a generalized equivalence relation for reasoning on continuous spaces, called a *distance metric*, $\delta : \mathcal{Z} \times \mathcal{Z} \to \mathbb{R}_{\geq 0}$. This has the following logical properties:

$$\frac{\delta(a,b) = 0}{a \equiv_\delta b} \qquad \frac{\delta(a,b)}{\delta(b,a)} \qquad \frac{a \qquad b \qquad c}{\delta(a,c) \leq \delta(a,b) + \delta(b,c)}$$

Definiteness *Symmetry* *Triangularity*

A *kernel function* $\Delta : (\mathcal{X} \to \mathcal{Y})^2 \to \mathbb{R}_{\geq 0}$ can be defined as a metric on $\mathcal{Z}$ with some additional structure. Specifically for every kernel function, there exists a feature map $\varphi : (\mathcal{X} \to \mathcal{Y}) \to \mathcal{Z}$ such that $\Delta : (f, g) \mapsto \langle \varphi(f), \varphi(g) \rangle$. Given a feature map $\boldsymbol{\varphi} : \mathbb{R}^m \to \mathbb{R}^n$, constructing a kernel corresponds to finding $\Delta \in \mathbb{R}^{m \times m}$ symmetric PSD, i.e. $\Delta^\top = \Delta$, and $0 \leq \mathbf{x}^\top \Delta \mathbf{x}$ for all $\mathbf{x} \in \mathbb{R}^m$ [34].

Consider the complexity of evaluating the inner product $\langle \boldsymbol{\varphi}(f), \boldsymbol{\varphi}(g) \rangle$ in feature space. The computational benefit of a kernel becomes apparent when $m \ll n$. Rather than applying $\boldsymbol{\varphi}$, then evaluating $\langle f^\top \boldsymbol{\varphi}, g^\top \boldsymbol{\varphi} \rangle$, we may apply $\Delta(f, g)$ directly for a $\Theta(m^2 - n^2)$ speedup. Known as the *kernel trick*, this shortcut may be easier to see categorically, where $f, g : \mathcal{X} \to \mathcal{Y}$.

$$\begin{array}{ccc} (\mathcal{X} \to \mathcal{Y})^2 & \xrightarrow{\varphi} & \mathcal{Z} \times \mathcal{Z} \\ & \searrow^{\Delta} & \downarrow{\scriptstyle \langle \cdot, \cdot \rangle} \\ & & \mathbb{R}_{\geq 0} \end{array}$$

By planting a single valid kernel Δ, one can grow a tree of kernel functions $\blacktriangle \subset (\mathcal{X} \to \mathcal{Y})^2 \to \mathbb{R}_{\geq 0}$ which may be described by the following grammar:

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

Some elementary kernel functions for various datatypes are given below:

	$\Delta(f, g)$	$\varphi(x)$
Polynomial	$(\mathbf{f}^\top \mathbf{g} + r)^q$	$\left[\sqrt{\binom{q}{\mathbf{n}} r^{n_0}} \prod_k x_k^{n_k}\right]^\top_{\mathbf{n} \in \{\mathbf{n} \mid \mathbf{1}^\top \mathbf{n} = q\}}$
Gaussian RBF	$e^{-\frac{\|\mathbf{f}-\mathbf{g}\|^2}{2\sigma^2}}$	$e^{-\gamma x^2}\left[1, \sqrt{\frac{(2\gamma)^i}{i!}} x^i\right]^\top_{i \in (0, \dim(x)-1]}$
Subset	$\prod_{i=1}^n (f_i g_i + 1)$	$[\varphi_{\text{POLY}}(x)_A]^\top_{A \subseteq [1,n]}$
Substring	$\sum_{\sigma \in \Sigma^*} (f * \sigma)(g * \sigma)$	$\lvert\{i \mid \sigma = x_{i..(i+\lvert\sigma\rvert)}\}\rvert$
Subtree [46]	$\Delta_{\text{WL}}(f, g)$	$\begin{cases} \delta_k(t, x) & t \stackrel{?}{=} x \\ \varphi_t(\overleftarrow{x}) + \varphi_t(\overrightarrow{x}) & \text{otherwise.} \end{cases}$
Subgraph [21]	$\Delta_{\text{SS}}(\varphi^{(h)}(f), \varphi^{(h)}(g))$	$\textsc{Hash}(\{\{\varphi^{(i-1)}(u) \forall u \in \mathcal{N}(x)\}\})$

An *approximate equivalence relation* is a binary relation $\mathcal{A} \subset (\mathcal{X} \to \mathcal{Y})^2$, e.g. $f \approx_d g \Leftrightarrow \Delta(f, g) \leq d$. Given a dataset $X \coloneqq [x^{(i)}]_{i=1}^n$, $Y \coloneqq [f(x^{(i)})]_{i=1}^n$ and a program $g : \mathcal{X} \to \mathcal{Y}$, we could use the extensional approach to evaluate $\Delta_X : (f, g) \mapsto \Delta_Y(Y, [g(x^{(i)})]_{i=0}^n)$ directly. Alternately, given an intensional descriptor $\varphi_\theta(\cdot) : (\mathcal{X} \to \mathcal{Y}) \to \mathcal{Z}$ we could construct a synthetic kernel $\Delta_\theta : (f, g) \mapsto \langle \varphi_\theta(f), \varphi_\theta(g) \rangle$, seeking $\theta^* = \text{argmin}_\theta (\Delta_\theta(f, g) - \Delta_X(f, g))^2$ over $f, g \overset{iid}{\sim} P(\mathcal{X} \to \mathcal{Y}), X \overset{iid}{\sim} P(\mathcal{X}^n)$ via gradient descent. The problem then becomes: how do we match the distribution of naturally-arising programs?

2.6 Probabilistic equivalence

So far, we have seen various notions of exact and approximate equivalence. However, what is needed is a way to quantify the probability a certain output is produced, or the likelihood of observing a representable function in the wild, given some prior knowledge. To do this properly, we need a language for propagating uncertainty through a computation. Probabilistic inference is one such framework for reasoning about uncertainty. In this section, we define a denotational and operational semantics for probabilistic inference.

Let E be a set of *events* and S a set of subsets of E. A *probability distribution* is a function $P : S \times \Sigma \to \mathbb{R}^+$, which satisfies the Kolmogorov [30] axioms:

(3) $P(S = s) \in \mathbb{R}^+, \forall s \in S$. ($S \stackrel{?}{=} s$ is usually written $S = s$ for brevity.)

(4) $P(E) \coloneqq 1$. ($P(S = E)$ may be shortened to $P(E)$ for brevity.)

(5) $P(X \cup Y) \coloneqq P(X) + P(Y), \forall X \cap Y = \emptyset$.

Given a distribution over a set X, we can *sample* from it to produce a single element from that set, a *random variable*:

$$\frac{\Gamma \vdash P(X) : X \times \Sigma \to \mathbb{R}^+ \qquad \Gamma \vdash x \sim P(X)}{\Gamma \vdash x : (X \times \Sigma \to \mathbb{R}^+) \rightsquigarrow X}\ \textsc{Sample}$$

The *joint*, $P(X, Y)$, is a distribution over the product of two sets, $X \times Y$:

$$\frac{\Gamma \vdash P(X) : X \times \Sigma \to \mathbb{R}^+ \qquad \Gamma \vdash P(Y) : Y \times \Sigma \to \mathbb{R}^+}{\Gamma \vdash P(X, Y) : X \times Y \times \Sigma \to \mathbb{R}^+}\ \textsc{Join}$$

Given a joint $P(X, Y)$, if we observe an event $y : Y$, this observation is called *conditioning* and the resulting distribution over X, a *conditional distribution*:

$$\frac{\Gamma \vdash P(X, Y) : X \times Y \times \Sigma \to \mathbb{R}^+ \qquad \Gamma \vdash y : Y}{\Gamma \vdash P(X \mid Y = y) : X \times \Sigma \to \mathbb{R}^+}\ \textsc{Cond}$$

In general, conditioning is order-dependent. Given a conditional distribution and its prior, to exchange the order of conditioning, we can use Bayes rule:

$$\frac{\overbrace{P(X \mid Y)}^{\text{Likelihood}} \quad \overbrace{P(Y)}^{\text{Prior}}}{\underbrace{P(Y \mid X)}_{\text{Normalize}} \propto \underbrace{P(X \mid Y)}_{\text{Observe}}\ \underbrace{P(Y)}_{\text{Sample}}}\ \textsc{Bayes}$$

When a conditional distribution $P(X \mid Y)$ does not depend on its prior, the events are said to be *independent*. Equivalently, if two distributions $P(X)$ and $P(Y)$ are multiplied to form a joint distribution $P(X, Y)$, we may also conclude that X and Y are independent events:

$$\frac{P(X \mid Y) \equiv P(X)}{X \perp Y}\ \textsc{Indep} \qquad \frac{P(X, Y) \equiv P(X)P(Y)}{X \perp Y}\ \textsc{Factor}$$

When two conditionals $P(X \mid Z)$, $P(Y \mid Z)$ are multiplied to form a joint distribution $P(X, Y \mid Z)$, X and Y are *conditionally independent given* Z:

$$\frac{P(X, Y \mid Z) \equiv P(X \mid Z)P(Y \mid Z)}{X \perp Y \mid Z}\ \textsc{CondIndep}$$

In order to sample from a univariate distribution, we can feed the output from a uniform PRNG into a *quantile function*. Unfortunately, most common distributions are defined in terms of their probability or cumulative density functions and have no representable quantile, so we are forced to approximate it by inverting the CDF, using e.g. Kolmogorov–Smirnov:

$$\frac{\frac{\texttt{x} \sim P(X) \qquad \texttt{CDF} : x \mapsto \int P(X = x)dx}{\texttt{PRNG() = CDF(x)}}\ \textsc{Draw}}{\texttt{x = INVCDF(PRNG())}}\ \textsc{Invert}$$

If we have two random variables sampled from known distributions and want to combine them, we must ensure the combination is also a probability distribution. Generally speaking, to combine two arbitrary RVs, we must integrate their density functions. For a dyadic function, we can take the double integral, which is known to be exchangeable under certain conditions [16]:

$$\frac{\frac{x \sim P(X) \qquad y \sim P(Y) \qquad z : X \times Y \to Z}{P(Z = z(x, y) \mid X = x, Y = y) = \int\int z(x, y)dxdy}}{\int_Y \int_X f(x, y)dxdy = \int_X \int_Y f(x, y)dydx}$$

While generally intractable in higher arity, these integrals can often be simplified by considering the specific dependence relation between the arguments. For instance, the sum of two independent RVs follows a distribution which can be obtained by convolving their respective density functions:

$$\frac{P(X) \qquad P(Y)}{P(X + Y) = P(X) * P(Y)}\ \oplus \qquad \frac{P(X) \qquad P(Y)}{P(X \times Y) = \int P(X)P(Y = \frac{x+y}{x})\frac{1}{|x|}dx}\ \otimes$$

More specifically, if we have a so-called *stable distribution* [31], evaluating the convolution can be obtained by combining their parameters μ and σ directly. As described in [51], there exist many simplified relations of the following variety, which can be discovered through analytic integration:

$$\frac{x \sim \mathcal{N}(\mu_x, \sigma_x^2) \qquad y \sim \mathcal{N}(\mu_y, \sigma_y^2) \qquad z = x + y}{z \sim \mathcal{N}(\mu_x + \mu_y, \sigma_x^2 + \sigma_x^2)}$$

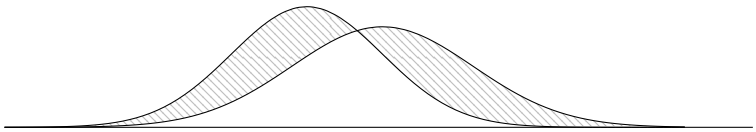
Given a joint distribution $P(X, Y)$, in order to remove a variable, we must sum or integrate over the conditional distribution. This procedure is known as *marginalization*, and the resulting distribution is called a *marginal*:

$$\frac{\Gamma \vdash P(X, Y) : X \times Y \times \Sigma \to \mathbb{R}^+}{\Gamma \vdash P(X) : X \times \Sigma \to \mathbb{R}^+ \propto \int P(X \mid Y = y) P(Y = y) dy} \textsc{Marg}$$

If we observe an event Y in a joint distribution where $0 < P(Y)$, we can divide or *normalize* by the prior to obtain the conditional distribution:

$$\frac{\Gamma \vdash P(X, Y) : X \times Y \times \Sigma \to \mathbb{R}^+}{\Gamma \vdash P(X \mid Y) : X \times \Sigma \to \mathbb{R}^+ = P(X, Y) \div P(Y)} \textsc{Cond}$$

It is seldom the case two distributions are exactly equal in practice. Given two distributions, we would like a way to quantify how similar they are.



Computing $\int |P_1(X) - P_2(X)| dx$ directly grows quickly intractable in higher dimensions. Various kernels, metrics and pseudometrics on multidimensional distributions have been proposed to alleviate this problem, for example, Jensen-Shannon, Kantorovich-Rubinstein, Kullback-Leibler, Earth-mover, et al. In general, approximations such as Gibbs sampling or Markov methods are usually required unless the integral can be decomposed into lower-dimensional sums and products of conditionally independent random variables (cf. § 3.2).

3 From computation to graphs

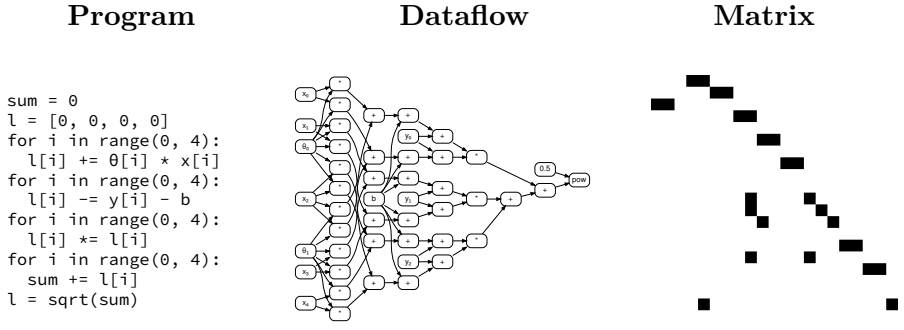
Graphs are algebraic structures [50] capable of representing a wide variety of procedural and relational information. A graph can be represented as a matrix $\{\mathbb{B}, \mathbb{N}, \mathbb{R}\}^{n \times n}$, with entries describing the presence, label, or distance between vertices. Linear algebra provides a unifying framework for studying many graph algorithms and program analysis tasks [28]. Graphs can also be defined algebraically, using an algebraic data type [13, 36], e.g.:

vertex	$\rightarrow$	int
adj	$\rightarrow$	[vertex]
context	$\rightarrow$	(adj, vertex, adj)
graph	$\rightarrow$	empty | context & graph

The graph sum emerges naturally, however there are multiple valid ways to define a graph product. As we will now see, this simple data structure can be used to describe many fundamental computational processes.

3.1 Programs are graphs

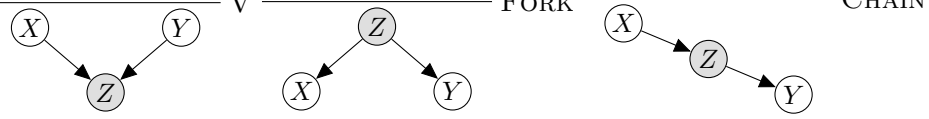
Programs are graphical structures with a rich denotational and operational semantics [22]. To effectively reason about programs in semantically similar but syntactically diverse settings requires models which incorporate features from the call graph [32] and surrounding typing context [2]. Many semantic structures, such as data and control flow [47] can be represented as a directed acyclic graph, which admits linear-time solutions to a variety of graph problems, including topological sorting, single-source shortest path and reachability queries. Many useful graph representations have been proposed in the literature, including call graphs, dataflow graphs, computation graphs [6], e-Graphs [52] down to boolean and arithmetic circuits [35].



3.2 Distributions are graphs

Probabilistic graphical models [27, 29] (PGMs) are a framework for probabilistic inference on distributions whose conditional independence structure may be expressed as a graph, with edges representing dependence relations between RVs. One type of PGM is a directed graphical model (DGM):

$$\frac{X \not\perp Y \mid Z}{\text{V}} \quad \text{V} \quad \frac{X \perp Y \mid Z}{\text{Fork}} \quad \text{Fork} \quad \frac{X \perp Y \mid Z}{\text{Chain}} \quad \text{Chain}$$

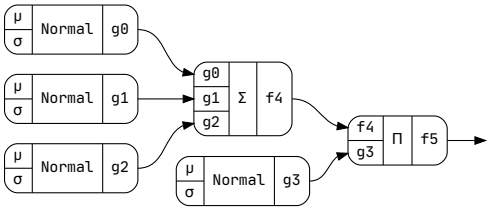


A Bayesian belief network (BN) is an acyclic DGM of the following form:

$$P(x_1, \ldots, x_D) = \prod_{i=1}^{D} P(x_i \mid \texttt{parents}(x_i))$$

PGMs are very expressive, but even approximate inference on BNs is known to be NP-hard [12]. We can faithfully represent a large class of PGMs and their corresponding distributions as probabilistic circuits (PCs) [9], which are capable of exact inference in polynomial time and empirically tractable to calibrate using SGD or EM. PCs share many algebraic properties in common with PGMs and can propagate statistical estimators like variance and higher moments using simple algebraic rules. A PC is formed by taking recursive sums and products of univariate distributions [15], as seen below:

$$PC \to v \sim \mathcal{D}$$
$$PC \to PC \oplus PC$$
$$PC \to PC \otimes PC$$



Given a BN, we can compile it to a SPN using the following procedure [7]:

```
procedure Translate(b: BN): SPN
    c ←VariableEliminate(b)
    s ←RedistributeParameters(c)
    s ←CompileMarginalized(s)
    return Canonicalize(s)
end procedure
```

```
procedure Canonicalize(s_0: SPN): SPN
    s_1 ← AddTerminals(s_0)
    s_1 ← MergeProducts(s_1)
    if s_0 = s_1 then return s_1
    else return Canonicalize(s_1)
end procedure
```

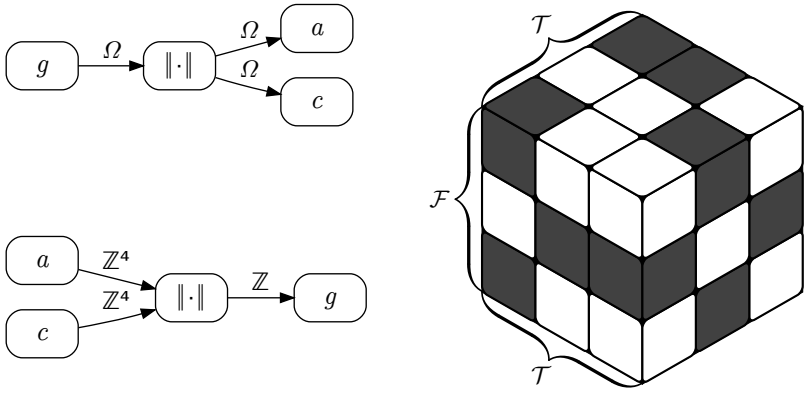
3.3 Knowledge is a graph

Abduction is the process of knowledge distillation: recognizing *patterns* or *motifs* in raw data, forming discrete *categories* or *types* of objects which share similar attributes, and adding edges representing abstract relations between types, such as composition, causation, or subsumption. The key idea is moving from the space of unstructured data to a structured domain containing abstract concepts and their relations. This domain is a graph.

Knowledge graphs [24] are *multi-relational graphs*, or *hypergraphs* whose edges possess at least one type. Our work is primarily concerned with *procedural knowledge*, or knowledge whose topology describes how to structure computation in order to achieve some desired task, although the technique we propose can also be applied to arbitrary directed, typed hypergraphs.

The vertices in our graph represent *functions* in some typed programming language and the edges represent their input and output types. Functions may be related by multiple types, and each type may relate multiple functions. Let $|\mathcal{T}|$ be the number of types in our system and $|\mathcal{F}|$ be the number of functions. We call our structure a *type tensor*, $\mathbb{B}^{|\mathcal{F}|\times|\mathcal{T}|\times|\mathcal{T}|}$, also known as a *logic table* or a *data cube*. Consider the case where $|\mathcal{F}| = |\mathcal{T}| = 3$:

Typed Graph **Type Tensor**



A type tensor indexed by a bijection $\mathcal{F} \times \mathcal{T} \times \mathcal{T} \to \mathbb{N}^3$ is a type-level function $C[\![\cdot]\!] \subseteq \mathcal{F}\times\mathcal{T} \to \mathcal{T}$ mapping a function $C : \mathcal{F}$ and type $T : \mathcal{T}$ to a set of reachable types $T^* \subseteq \mathcal{T}$, the congruence closure $C[\![T^*]\!] \equiv T^*$ of T under C. Given a concrete type $S : \mathcal{T}$ and a one-hole context, $C[\![\cdot]\!] : ? \to S$, synthesis in our type system consists of applying tensor contraction to compute the preimage of S under C, $C^{-1}[\![S^*]\!]$. We leave this procedure for future work.

3.4 Algebraic graphs

The connection between algebra and graphs runs deep, unifying many seemingly disparate topics. A commutative monoid $(S, \bullet, ①)$ is a set S with a binary operator $\bullet : S \times S \to S$ which has the following properties:

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c} \qquad \frac{a \bullet ①}{a} \qquad \frac{a \bullet b}{b \bullet a}$$

Associativity *Neutrality* *Commutativity*

A semiring algebra, denoted $(S, \oplus, \otimes, ⓪, ①)$, is a set together with two binary operators $\oplus, \otimes : S \times S \to S$ such that $(S, \oplus, ⓪)$ is a commutative monoid and $(S, \otimes, ①)$ is a monoid. It has the following additional properties:

$$\frac{a \otimes (b \oplus c)}{(a \otimes b) \oplus (a \otimes c)} \qquad \frac{(a \oplus b) \otimes c}{(a \otimes c) \oplus (b \otimes c)} \qquad \frac{a \otimes ⓪}{⓪}$$

Distributivity *Annihilation*

Using iterated matrix multiplication on semirings, it is possible solve a wide variety of path problems. Known as *propagation* or *message passing*, this procedure consists of two steps: *aggregate* and *accumulate*. Let $\mathbf{D}_{st}$ denote some distance metric on a path between two vertices s and t in a graph. To obtain $\mathbf{D}_{st}$, one may run the following procedure on a desired path algebra:

$$\mathbf{D}_{st} = \overbrace{\bigoplus_{P \in P^*_{st}} \underbrace{\bigotimes_{e \in P} W_e}_{\text{Aggregate}}}^{\text{Accumulate}}$$

S	$\oplus$	$\otimes$	⓪	①	Path
$\mathbb{R} \cup \{\infty\}$	min	$+$	∞	0	Shortest
$\mathbb{R} \cup \{\infty\}$	max	$+$	$-\infty$	0	Longest
$\mathbb{R} \cup \{\infty\}$	max	min	0	∞	Widest

Many dynamic programming algorithms, including Bellman-Ford, Floyd-Warshall, Dijkstra, error, belief and expectation propagation, Markov chains, and others can all be neatly expressed as message passing with a semiring algebra. We refer the curious reader to explore [19, 4] for a more complete summary of the algebraic path problem and its many wonderful applications.

4 From code to procedural knowledge

Procedural knowledge is knowledge designed or discovered to facilitate a process. Unlike *relational knowledge* which may be bidirectional, procedural knowledge has a specific direction or goal. The methods presented in this literature review have broad applications to parsing and indexing procedural knowledge. Given a sequence of *paths*, or *traces* representing discrete steps in a goal-directed process with intermediate results, our research seeks to provide a mechanism for estimating procedural similarity, to synthesize or recommend semantically similar procedures under feasible reorderings. Specifically, our work focuses on the domain of *programming* knowledge.

Code shares many statistical properties in common with natural language and can be studied using natural language properties [23]. Unlike natural language, all syntactically-valid code has an unambiguous grammar. Furthermore, code primarily consists of *functions* or *procedures* intended to operate a machine. The underlying programming model may be relational, object-oriented or functional, but all mainstream programming languages support some form of *procedure* which accepts inputs and produces outputs.

All procedures in a statically-typed programming language have a *type signature*, for example, `getHomePhone: Person → PhoneNumber`. This represents a *contract* between the procedure and its caller: if the procedure `getHomePhone` is called with a `Person`, it will return a `PhoneNumber`. All other inputs and outputs are forbidden by the type system. While often beneficial to consider more narrow definitions such as soundness or correctness, for procedural similarity these assumptions are unnecessary.

Application programming interfaces (APIs) are a set of interfaces which describe available ways of structuring computation to achieve a set of related programming tasks. We call the graph of all possible ways to compose an API the *API surface*. One traverses the API surface by composing accessible procedures in a *call graph*. Thus, we can view the API as kind of a *procedural knowledge base* representing common data transformations and how to compose them. In practice, how to achieve some desired goal is often far from obvious, requiring a large amount of documentation to explain.

Many consumers of popular APIs publish code and documentation in open source repositories, a largely untapped source of knowledge for programming tools. Our work seeks to find ways of linking knowledge contained in open source repositories to help users locate examples and compose software applications. In the following section, we propose three applications of procedural pattern recognition for documentation alignment (§ 4.1), code search (§ 4.2), program repair (§4.2), and eDSL generation (§ 4.2).

4.1 Documentation alignment

Documentation is an indispensable resource for software developers learning to use a new API or programming language. Maintainers of popular software projects often publish web-based developer documents, typically in markup languages such as HTML or Markdown. These documents contain a mixture of natural language sentences, code snippets, and hyperlinks to related documents and source code files. All of these artifacts hold rich semantic information: the markup graph describes the text in relation to other entities in the document hierarchy, while the link graph describes relationships between relevant documents or artifacts in a software repository.

Consider the typical workflow of a software developer who is seeking information about an unfamiliar API. To effectively locate relevant documentation, she must first copy a specific fragment of text (e.g. a function name, error message, or identifier) from a development environment into a search engine, providing relevant contextual information. The query must be descriptive enough to retrieve relevant documents with high probability, while omitting extraneous information (e.g. user-defined tokens) unlikely to occur outside the scope of the developer's personal environment or project.

Prior work in information retrieval for software development investigated recommending API documentation [45] and Q&A content [48] to developers. Similar work in natural language processing has studied the relationship between comments and source code entities [25, 39] strictly within source code. Examples of cross-domain entity linking in the source-to-doc (S2D) and doc-to-source (D2S) setting are scarce, however these results indicate alignment between natural language and software artifacts may be feasible.

Our work seeks to facilitate procedural knowledge discovery by enriching lexical queries with semantic information extracted from a programming environment, and prioritizing semantically relevant software artifacts among a set of matching search results. Broadly, the tools we have proposed in this literature review can be used to study both source code and procedural knowledge graphs, however due to the paucity of cross-domain entity links between code and natural language artifacts, reasoning about cross-domain relations will require developing new approaches to feature engineering, unsupervised learning and entity alignment in the low-data regime.

Such an application, if successful, would allow developers to more quickly and easily locate semantically or contextually relevant code samples in API documentation and open source repositories. The same tools could also help authors maintain a consistent set of API documentation and usage examples across a large codebase, a persistent obstacle when evolving any API.

4.2 Code search

Given a learned similarity metric between procedures, one straightforward application is code search. Prior work in this area has explored type-directed [26], learning-based [20] and semantic search [42] techniques. These techniques all use a fixed, or synthetic ordering over search results. For a given hole context, there are often many valid completions within an API or codebase. Given a corpus of procedures in their surrounding typing environment, is it possible to estimate a probability distribution on a shared embedding between contexts and results, and measure the likelihood that a given search result occurs in an empty location? This requires:

1. Efficiently searching a corpus for a well-typed pattern
2. Ranking the matching search results by semantic alignment
3. Incorporating information into user's context (e.g. variable renaming)

Given a cursor and the surrounding context (e.g. in an IDE or editor), such a tool would need to search a database for the most similar contexts, extract common snippets to estimate their *concordance* or *agreement* with the surrounding code context, then adapt the foreign code snippet into to the user's context. External contexts including public code samples or API-documentation (e.g. fixes or repairs for compiler error messages) could help the user or complete some unfinished piece of code. Some open research questions include:

1. **Semantic segmentation**: How do we slice or truncate context?
2. **Graph search**: What kernel be used for fast subgraph detection?
3. **Context ranking**: What features to use to best measure similarity?
4. **Refactoring**: How to integrate a selected result into the user's code?

The proposed method presents an interesting engineering challenge. To work for code completion, it would need to have relatively low latency for a pleasant user experience. While the model could be trained on a large corpus offline, the contextual embedding would need to be periodically recomputed as the surrounding typing environment changes in the editor. Furthermore, the search and retrieval speed would need keystroke latency to be effective.

4.3 Program synthesis

In this proposal, we have seen a number of procedures for generating, or synthesizing a short procedure from a space of valid procedures. Various approaches are presented for determining the equivalence of valid procedures and sampling from a constrained space of representable functions to produce a higher-order probabilistic function, satisfying a set of constraints.

Our initial approach described in §3.3, simply enumerates paths between types in our type system, but does not tell us how likely a given path is to occur in a codebase. To compute this probability, we need a stochastic transition matrix between functions, $\mathcal{F} \times \mathcal{F}$ representing the probability that some function is to be composed with another. Given a grammar and type system, we would like a way of sampling novel programs according to the distribution of programs which occur in some codebase.

4.4 DSL generation

Most knowledge starts with pen on paper. It is generally possible for developers to translate well-structured pseudocode into code. However, there are many valid translations – the same algorithm implemented in the same language by different authors is seldom written the same way. Instead of translating these ideas from scratch, it may be possible to encode just a few axioms and enough knowledge to derive a family of algorithms, reusing existing optimizations written by other developers and selecting the most appropriate procedure for computing a desired quantity, e.g. using a sufficiently expressive logic system, and a database of simple facts and relations.

Knowledge systems or *ontologies* are a collection of related facts designed or discovered by human beings. For example, we can treat mathematical knowledge as a kind of symbolic rewrite system. This has been successfully operationalized in Theano [5] and other DSLs. It is possible to view constructive mathematics libraries like Metamath [33], Rubi [44], Arend, KMath [38] and Kotlin∇ [11] as also working towards this high-level goal.

In knowledge engineering, approximate equality is known as entity *alignment* or *matching*. With an approximate matching algorithm, we could accurately detect near-duplicates in a large codebase, find similar documentation snippets, retrieve contextually or semantically relevant code samples to assist developers writing unfamiliar code, and search for bugs in code or fixes from an external knowledge base to repair them. All of these tasks require some notion of semantic similarity between procedures.

5 Acknowledgements

The author wishes to thank his advisor Jin Guo, Xujie Si and Fuyuan Lyu, for providing several helpful comments on this proposal and Torsten Scholak, David Yu-Tung Hui, Disha Shrivastava, Jordi Armengol-Estapé, Matt Rice, Dave Keenan, and John Tromp for various miscellaneous feedback.

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] John S Baras and George Theodorakopoulos. Path problems in networks. *Synthesis Lectures on Communication Networks*, 3(1):1–77, 2010.

[5] 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.

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

[7] Cory J Butz, Jhonatan S Oliveira, and Robert Peharz. Sum-product network decompilation. *arXiv preprint arXiv:1912.10092*, 2019.

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

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

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

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

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

[13] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

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

[15] Abram Friesen and Pedro Domingos. The sum-product theorem: A foundation for learning tractable models. In *International Conference on Machine Learning*, pages 1909–1918, 2016.

[16] Guido Fubini. Sugli integrali multipli. *Rend. Acc. Naz. Lincei*, 16:608–614, 1907.

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

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

[19] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[20] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[21] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[22] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted

symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

[23] 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.

[24] 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.

[25] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[26] 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.

[27] Michael I Jordan. An introduction to probabilistic graphical models, 2003.

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

[29] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[30] Andrei Nikolaevich Kolmogorov. Grundbegriffe der Wahrscheinlichkeitsrechnung, 1933.

[31] Paul Lévy. Calcul des probabilités. 1925.

[32] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. A neural-network based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, Internetware '19, New York, NY, USA, 2019. Association for Computing Machinery.

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

[34] James Mercer. Functions of positive and negative type, and their connection the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character*, 209(441-458):415–446, 1909.

[35] 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.

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

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

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

[39] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. Associating natural language comment and source code entities. In *AAAI*, 2020.

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

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

[42] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *PLDI*, pages 1066–1082, 2020.

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

[44] 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.

[45] Martin P Robillard and Yam B Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[46] Nino Shervashidze and Karsten Borgwardt. Fast subtree kernels on graphs. *Advances in neural information processing systems*, 22:1660–1668, 2009.

[47] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pages 7751–7762, 2018.

[48] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 392–403, New York, NY, USA, 2016. Association for Computing Machinery.

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

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

[51] Brandon T Willard. miniKanren as a tool for symbolic computation in python. *arXiv preprint arXiv:2005.11644*, 2020.

[52] 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.