Pattern Recognition in Procedural Knowledge

Breandan Considine

January 17, 2021

Contents

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 [19], code has a well-defined grammar and unambiguous semantics [35]. We can use these properties to precisely reason about operational or procedural correctness.

Prior work explored differentiable programming [9]. 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 [13], and simulate or execute them on various input data [7] 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 [34], 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 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 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:

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

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:

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

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 δ_k nor $\nabla \delta_d$ are a satisfactory basis for logical equality. To allow a more flexible definition of the $\overset{?}{=}$ 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:

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

$$\textit{Identity} \qquad \textit{Symmetry} \qquad \textit{Transitivity} \qquad \textit{Congruence}$$

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 [16] and higher order theories [17]. We know there can be no machine which accepts every equality and rejects every disequality in a universal language [40]. By extension, any nontrivial property of partial functions is undecidable [36].

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.

4

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 $n \in \mathbb{N}$.

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

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

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, $\text{EQ}[f, g]$ can be established as follows:

$$\frac{\begin{array}{c} \dfrac{f := abc, \ g := cba \in \Gamma_g^0}{\Gamma_g^0 \vdash \{abc\}, \{(abc, abc)\}} \text{ INIT} \\ \dfrac{}{\Gamma_g^1 \vdash \{\dots, bac, acb\}, \{\dots, (abc, bac), (abc, acb)\}} \text{ SUB} \\ \dfrac{}{\Gamma_g^2 \vdash \{\dots, bca, cab\}, \{\dots, (bac, bac), (acb, acb), (bac, bca), (acb, cab)\}} \text{ SUB} \\ \dfrac{}{\Gamma_g^3 \vdash \{\dots, \mathbf{cba}\}, \{\dots, (bca, bca), (cab, cab), (cab, \mathbf{cba})\}} \text{ SUB} \end{array}}{\Gamma_g^3 \vdash f \equiv_\Omega g \text{ by } G^{-3}(g := cba) = \{f := abc\}} \text{ EQ}$$

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 $\text{EQ}[f, g], \forall f \equiv_\Omega g \in \mathcal{X} \to \mathcal{Y}$. It can be shown the above procedure requires:

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

Assuming termination, $\mathcal{O}_{\text{EQ}} = \Theta_{\text{NEQ}}$ although $\mathbb{E}[\Theta_{\text{EQ}} | f \equiv g]$ 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.

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

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

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 $\Omega := \{(a, 1), (b, 2), (c, 2)\}$:

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

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* [32] 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 $C[\![t]\!] \Uparrow$.

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

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

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:

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

$$\textit{Definiteness} \qquad \textit{Symmetry} \qquad \textit{Triangularity}$$

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 $\mathbf{x} \in \mathbb{R}^m$ [29].

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 $f, g : \mathcal{X} \to \mathcal{Y}$.

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

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:

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

Some elementary kernel functions for various datatypes are given below:

	$\Delta(f, g)$	$\varphi(x)$
Polynomial	$\left(\mathbf{f}^\top \mathbf{g} + r \right)^q$	$\left[\sqrt{\binom{q}{\mathbf{n}}} r^{n_0} \prod_k x_k^{n_k} \right]^\top_{\mathbf{n} \in \{\mathbf{n} \mid \mathbf{1}^\top \mathbf{n} = q\}}$
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	$\sum_{\sigma \in \Sigma^*} (f * \sigma)(g * \sigma)$	$\left| \{ i \mid \sigma = x_{i..(i+|\sigma|)} \} \right|$
Subtree [38]	$\Delta_{\text{WL}}(f, g)$	$\begin{cases} \delta_k(t, x) & t \overset{?}{=} x \\ \varphi_t(\overleftarrow{x}) + \varphi_t(\overrightarrow{x}) & \text{otherwise.} \end{cases}$
Subgraph [18]	$\Delta_{\text{SS}}(\varphi^{(h)}(f), \varphi^{(h)}(g))$	$\text{HASH}(\{\{\varphi^{(i-1)}(u) \forall u \in \mathcal{N}(x)\}\})$

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 $X := [x^{(i)}]_{i=1}^n$, $Y := [f(x^{(i)})]_{i=1}^n$ 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 $\theta^* = \operatorname{argmin}_\theta (\Delta_\theta(f, g) - \Delta_X(f, g))^2$ 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 [26] axioms:

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

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

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

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

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

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

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

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*:

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

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

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

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:

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

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*:

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

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:

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

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 [15]:

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

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:

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

More specifically, if we have a so-called *stable distribution* [27], evaluating the convolution can be obtained by combining their parameters μ and σ directly. As described in [42], there exist many simplified relations of the following variety, which can be discovered by 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*:

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

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

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

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.
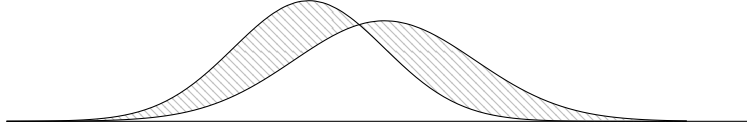
Computing $\int |P_1(X) - P_2(X)| dx$ directly grows quickly intractable in higher dimensions. Various kernels, metrics and pseudometrics on distributions have been proposed to alleviate this problem, for example: Kantorovich-Rubinstein, Kullback-Leibler, Lévy-Prokhorov, Jensen-Shannon, EMD, et al.

3 From computation to graphs

Graphs are algebraic structures [41] 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 weight between vertices. The language of linear algebra provides a unifying framework for studying many graph algorithms and program analysis tasks [24]. and sparse matrix algorithms enable efficient processing of large graphs on modern graphics processors [23].

Graphs can also be defined inductively, using an algebraic data type [31].

vertex → int

adj → [vertex]

context → (adj, vertex, adj)

graph → empty | context graph

The sum emerges naturally, however there are multiple ways to define the product of two graphs. Semirings arise in strange and marvelous places. $(min, +), (max, \times)$

3.1 Programs are graphs

Programs are graphical structures at multiple levels of abstraction. Call graphs, dataflow graphs, computation graphs [6], e-Graphs [43] in reasoning systems, all the way down to boolean circuits [30] in numerical computing.
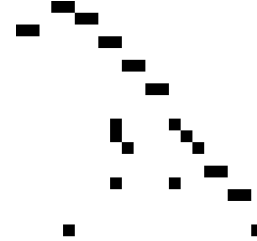
Program	Dataflow	Matrix


```
sum = 0
l = [0, 0, 0, 0]
for i in range(0, 4):
  l[i] += θ[i] * x[i]
for i in range(0, 4):
  l[i] -= y[i] - b
for i in range(0, 4):
  l[i] *= l[i]
for i in range(0, 4):
  sum += l[i]
l = sqrt(sum)
```
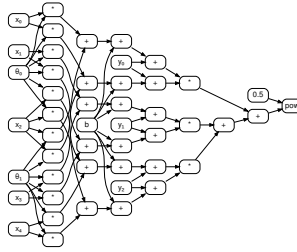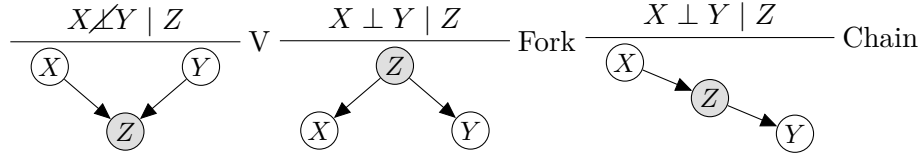
3.2 Distributions are graphs

Probabilistic graphical models [22, 25] (PGMs) are a framework for probabilistic inference on distributions whose conditional dependence structure may be expressed as a graph, whose edges capture conditional independence relations between RVs. One type of PGM is a directed graphical model (DGM).

$$\frac{X \not\perp Y \mid Z}{\begin{array}{c} X \rightarrow Z \leftarrow Y \end{array}} \text{V} \quad \frac{X \perp Y \mid Z}{\begin{array}{c} Z,\ X \leftarrow Z \rightarrow Y \end{array}} \text{Fork} \quad \frac{X \perp Y \mid Z}{\begin{array}{c} X \rightarrow Z \rightarrow Y \end{array}} \text{Chain}$$

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

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

Probabilistic graphical models (PGMs) are very expressive, but even approximate inference on belief networks (BNs) is NP-hard [11] We can faithfully represent a large class of PGMs and their corresponding distributions as probabilistic circuits (PCs) [8], 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.

Recent work [8] has explored tractable models for probabilistic reasoning based on semiring algebras. Semirings are known to have many useful applications in graph theory [12] and formal languages [5]. A semiring algebra has two operators, $\oplus$ and $\otimes$, with the usual properties. In particular, distributivity holds.

$$\frac{X \otimes (Y \oplus Z)}{(X \otimes Y) \oplus (X \otimes Z)} \quad \frac{(Y \oplus Z) \otimes X}{(X \otimes Y) \oplus (X \otimes Z)} \text{Distrib}$$

The sum product network (SPN) is a commutative semiring over univariate distributions [14]:

$$PC \rightarrow v \sim \mathcal{D} \quad PC \rightarrow PC \oplus PC \quad PC \rightarrow PC \otimes PC$$

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 [20] 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**
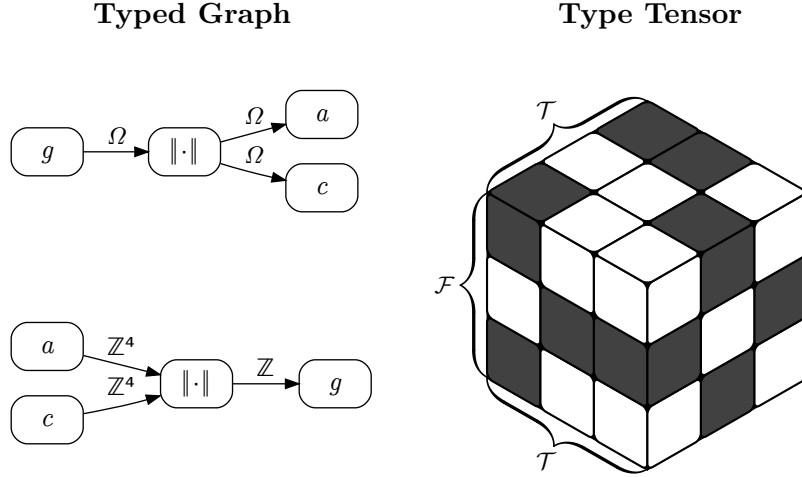
A type tensor indexed by a bijection $\mathcal{F} \times \mathcal{T} \times \mathcal{T} \to \mathbb{N}^3$ is a type-level function $C[\![\cdot]\!] \subseteq \mathcal{F} \times \mathcal{T} \to \mathcal{T}$ mapping a function $C : \mathcal{F}$ and type $T : \mathcal{T}$ to a set of reachable types $T^* \subseteq \mathcal{T}$, 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 a series of tensor contractions computing the preimage of S under C, $C^{-1}[\![S]\!]$. While subtyping, parametric polymorphism

and higher-kinded types are possible to encode by considering higher-rank tensors, here we devote our attention to the simply-typed, rank-3 setting, in which every function maps a concrete type to a concrete type closed under composition. The typing rules for this procedure are given below.

3.4 Message passing algorithms

Many algorithms can be implemented as message passing on graphs...

4 From code to procedural knowledge

Experts typically encode their knowledge using pen and paper. It is generally possible for skilled coders to translate textual information into code. However 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. 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.

One way to think of big code is as a procedural knowledge system, describing common data transformations and how to achieve them. Suppose we had a *bibliotheca universalis* containing many human-written code examples, which a human could select from manually (a la Hoogle [21]) or which could be linked to during compilation. 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∇ [10] and other DSLs. More broadly, we can also think of constructive mathematics libraries like Metamath [28], Rubi [37], Probonto [39] and KMath [33] 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 graph 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 k 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. 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.

17

5 Acknowledgements

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] Jean-Philippe Bernardy and Koen Claessen. Efficient divide-and-conquer parsing of practical context-free languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 111–122, New York, NY, USA, 2013. Association for Computing Machinery.

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

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

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

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

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

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

[12] Stephen Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 101–110, 2013.

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

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

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

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

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

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

[29] 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.

[30] 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.

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

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

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

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

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

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

[37] 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.

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

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

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

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

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

[43] 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.