

NFT Marketplace

Brian, Amy, & Andrew

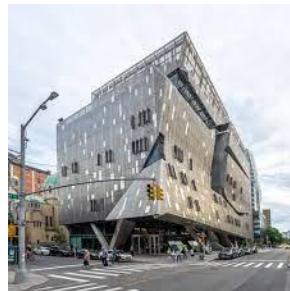
Background

- Art graduates struggle to earn a living following graduation due to lack of reputation & startup funds.
- Students are unable to present art in fairs, because strict entry deadlines conflict with school work.
- Art market is expanding into the digital space with the emergence of NFTs & Smart Contracts.

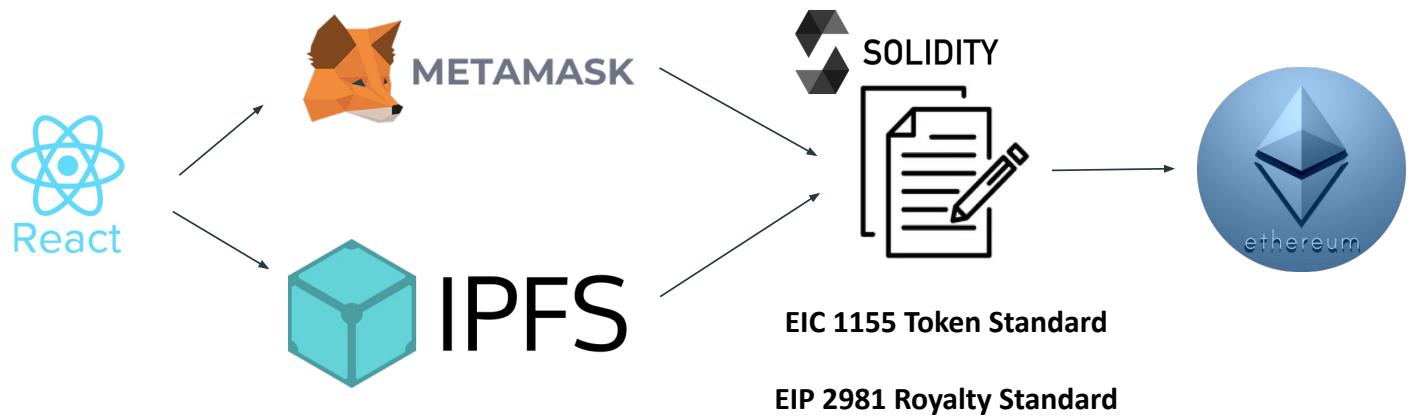


Solution / Objective

- Ethereum-based NFT marketplace that introduces students to their respective markets using the Cooper Union name brand.
- Restrict minting to Cooper Union students alone to give students a platform for their work to be seen and purchased.
- Proof-of-concept to display possibilities of applying an NFT marketplace to an academic institution and initiate conversations within Cooper Union.

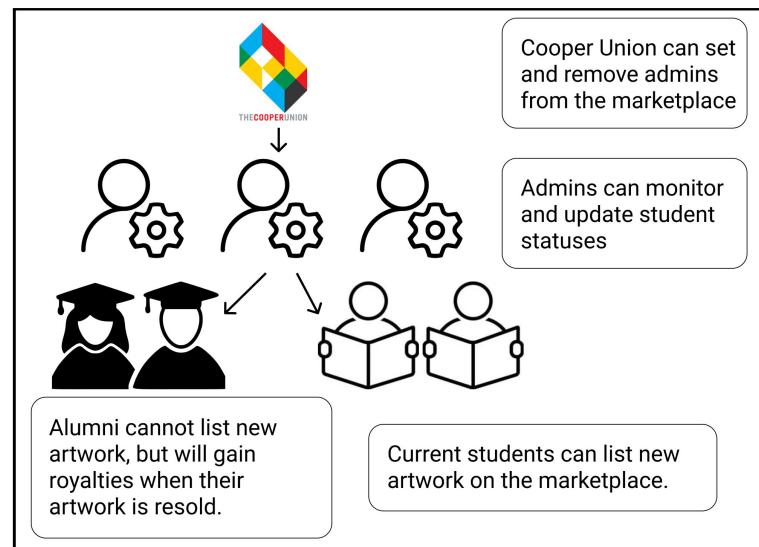


Technologies Used



Marketplace Hierarchy

- Marketplace roles are built into the smart contracts and tied to Ethereum wallet addresses.



Demo (Home Page)

The screenshot shows the homepage of the CU Marketplace. At the top, there is a navigation bar with links for "CU Marketplace", "About", "Explore", "Create", and a user icon. Below the navigation bar, a main banner features the text "For the Advancement of Science and Art" and "Explore the Cooper Union Student-Made NFT Collection". A blue button labeled "Discover New Art" is centered below the banner. The main content area has a purple header titled "Recent Listing" with two thumbnail images of buildings and a colorful geometric logo for "THE COOPER UNION". Below this, a section titled "How It Works" is divided into three sections: "Connect Your MetaMask Wallet" (with a wallet icon), "Create NFTs from your artwork" (with a camera icon), and "List your NFTs on the marketplace" (with a storefront icon). There are also "Student" and "Buyer" tabs above the "How It Works" section.

CU Marketplace

About Explore Create

For the Advancement of Science and Art

Explore the Cooper Union Student-Made NFT Collection

Discover New Art

Recent Listing

THE COOPER UNION

How It Works

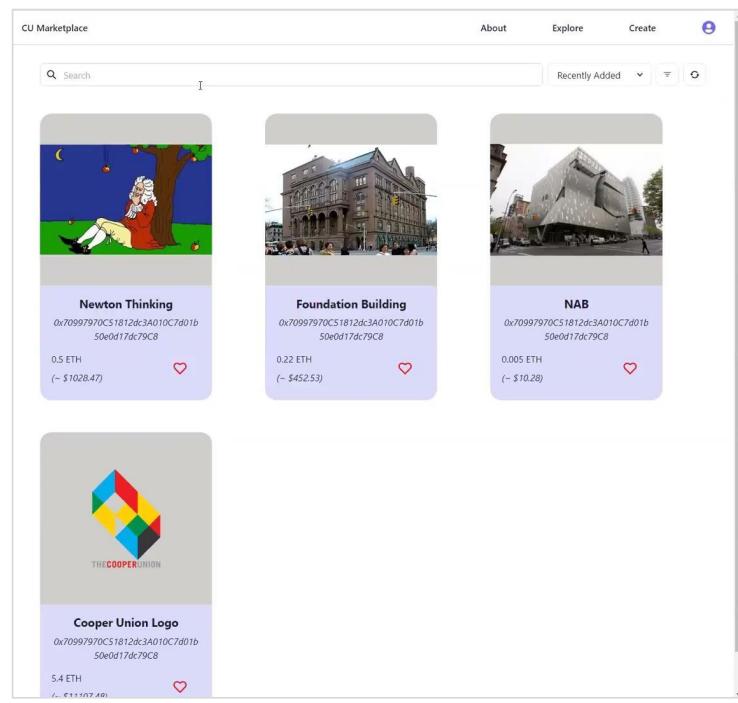
Student Buyer

Connect Your MetaMask Wallet

Create NFTs from your artwork

List your NFTs on the marketplace

Demo (Explore Page)



Demo (NFT Page)

CU Marketplace

About Explore Create 



NAB

Picture of Cooper Union NAB Building

Listed By: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8

Sale Price: 0.005 ETH (~ \$10.31)

[Cancel Listing](#) [Edit Listing](#)

Demo (Sign in with MetaMask)

CU Marketplace About Explore Create [Sign In](#)



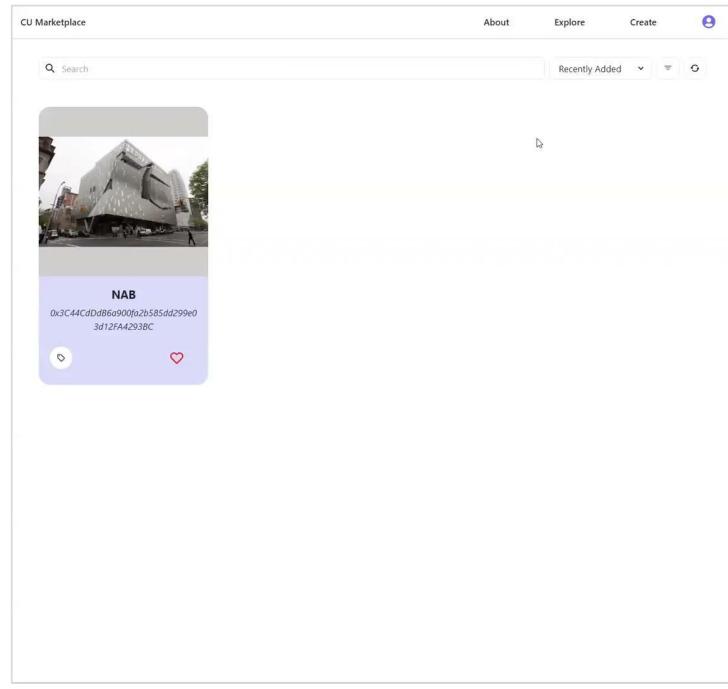
NAB

Picture of Cooper Union NAB Building

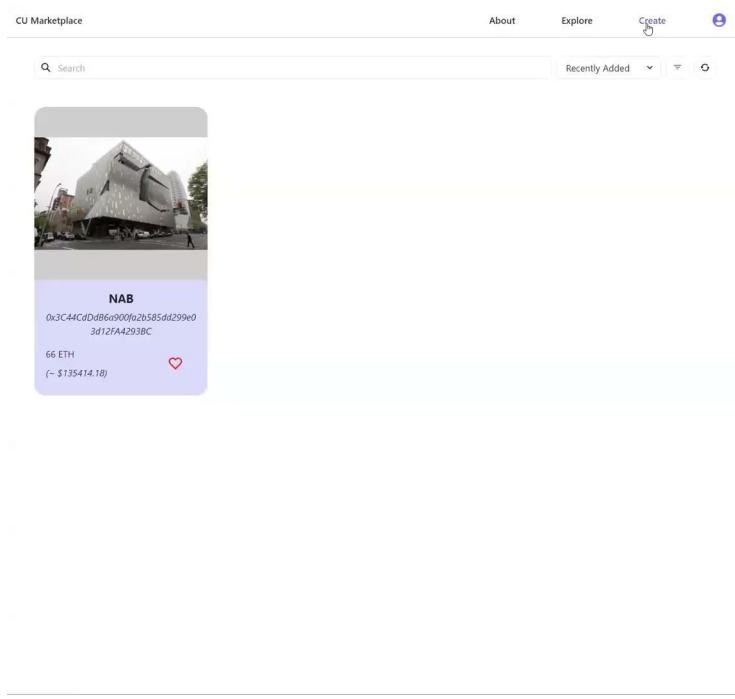
Owner: 0x3C44CdD6a900fa2b585dd299e03d12FA4293BC

↓

Demo (My NFTs Page)



Demo (NFT Creation - Not a Student)



Demo (Update Student's Status)

The screenshot shows a web page from the CU Marketplace. At the top, there is a navigation bar with links for "About", "Explore", "Create", and a user icon. Below the navigation, a large heading reads "Create a unique piece of digital artwork". Underneath this, a sub-headline says "Turn your art into a one-of-a-kind NFT". A note below states: "You must be a current student to mint NFTs. If you believe you should have access, click the button below." A blue "Request Access" button is centered at the bottom of this section.

Demo (NFT Creation)

Create a unique piece of digital artwork

Turn your art into a one-of-a-kind NFT

NFT Name *

NFT Description *

What are royalties?

Disable royalties for this NFT

Royalty Percentage * ⓘ

Royalty Recipient *

List NFT on Marketplace

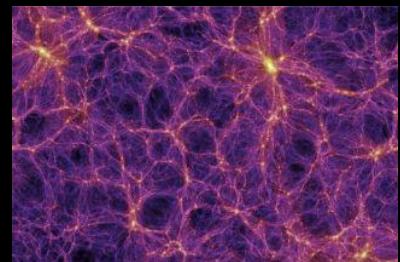
Price *

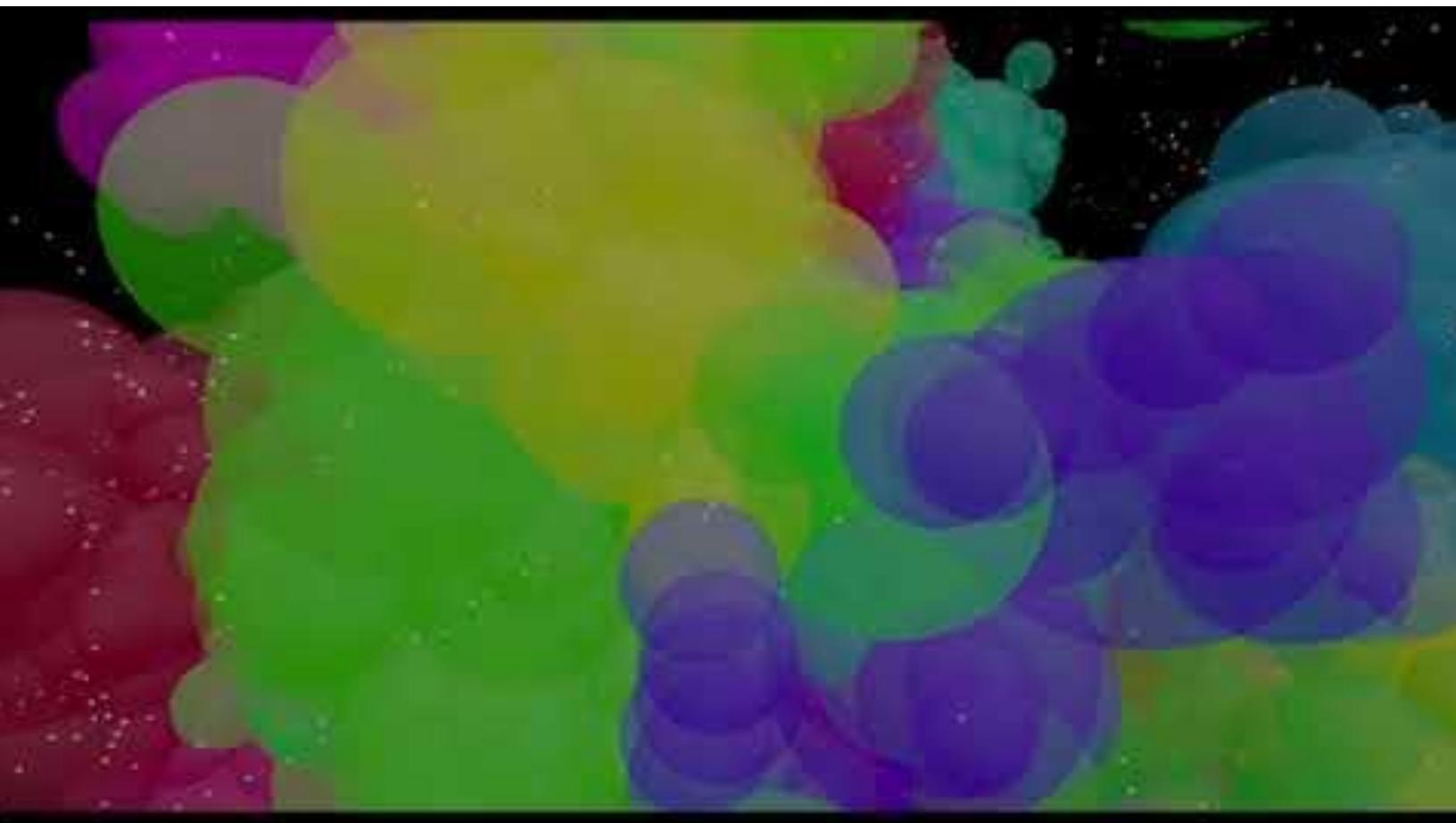
Future Work

- Implement system where Cooper allots an allowance of ETH to each student for gas fees (to promote equity in student listing opportunities).
- Allow alumni to donate NFTs for Cooper to sell on marketplace.
- Add alternate sale methods (i.e auction).
- Research eco-friendly blockchains for final product.

Machine Learning And Data Visualization for Cosmic Voids

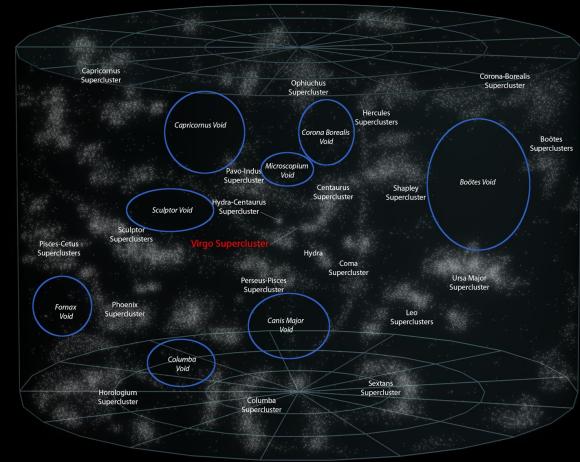
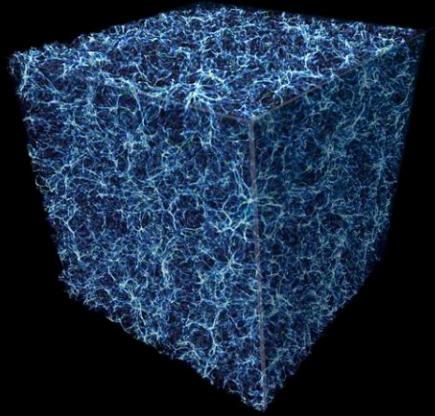
Bonny Wang
Allister Liu
Prof. Pisani
Prof. Keene





What are Voids?

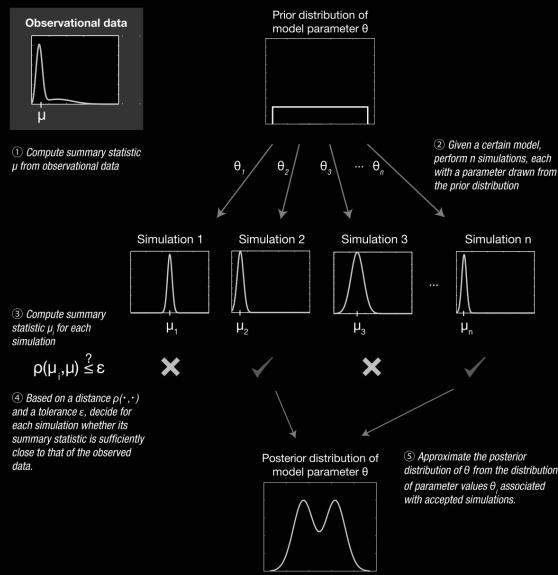
- The large under-dense regions in the Universe dominated by dark energy
- A novel probe for our understanding of the Universe
- Yet an understudied field



Problem Statement

- The intractability of the likelihood function in astrophysical simulations
- Computation

Traditional Method for Scientific Inference



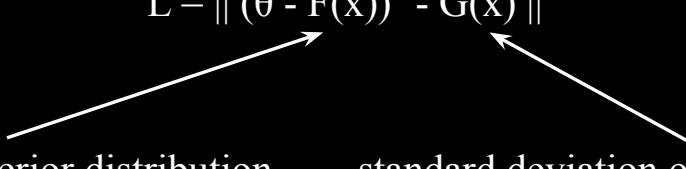
- The likelihood function is intractable in complex simulations
- Approximate Bayesian computation is time-consuming and computationally expensive
- The exact value of the posterior probability is unknown and may not be normalized



Density Estimation-based
Likelihood-Free Inference

Moment Networks

- A simple hierarchy of fast neural regression models
- Addresses the “**curse of dimensionality**”
- Directly skipping to the estimations of mean, standard deviation of the predicted parameter

$$L = \| (\theta - F(x))^2 - G(x) \|^2$$


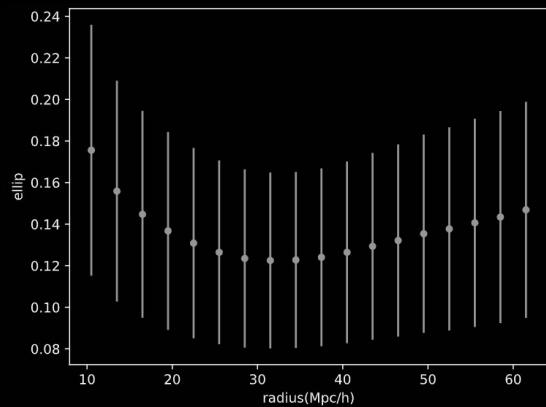
mean of posterior distribution standard deviation of posterior distribution

Dataset

GIGANTES

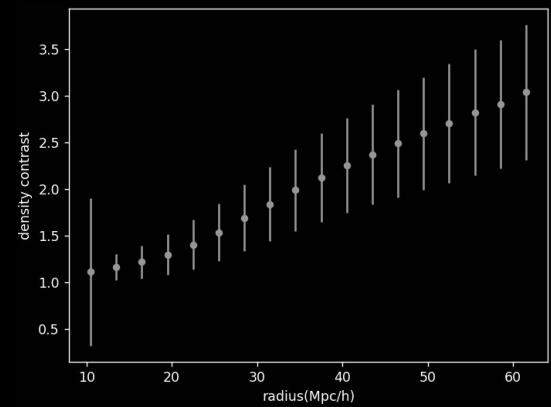
- An extensive and realistic void catalog
- Contains over 1 billion cosmic voids with their different cosmological parameters – over 20 TB
- Created by running the void finder algorithm VIDE on simulations

Relationship between what we want to explore and the previous study



Void Ellipticity:

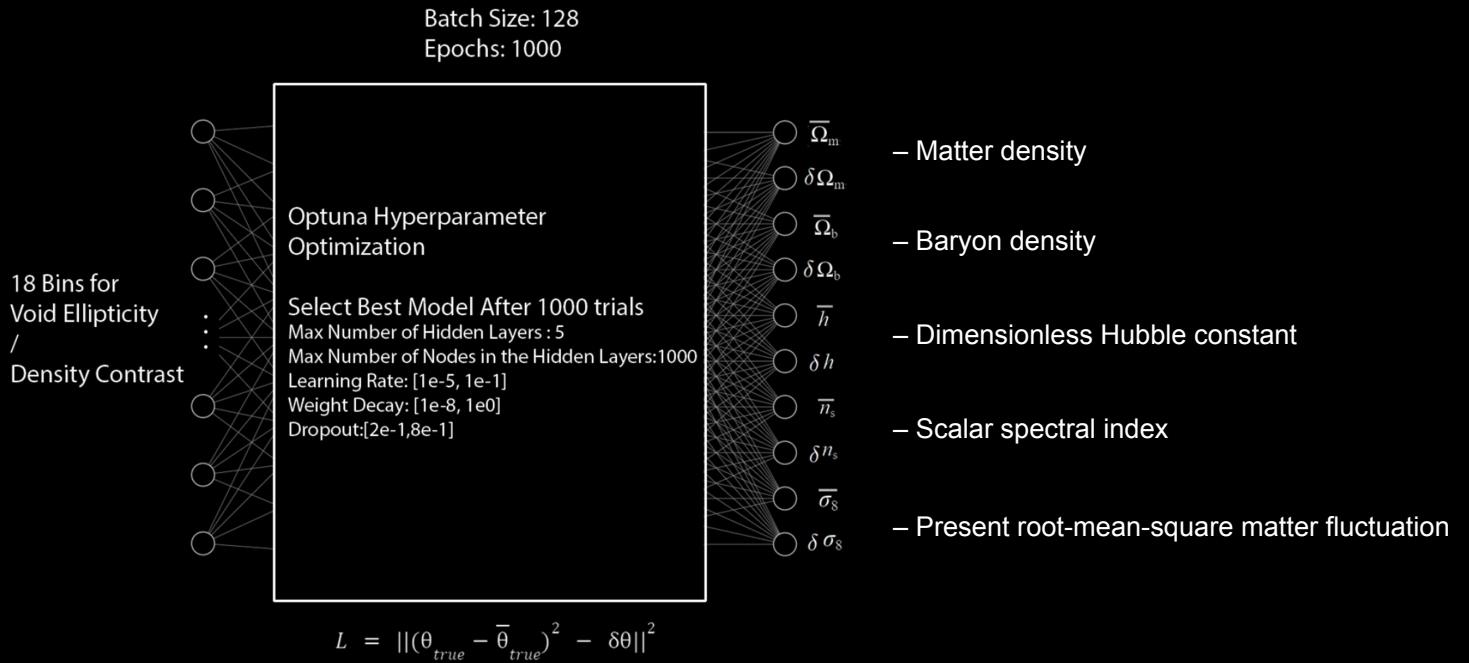
Describes the elliptical shape of a void



Void Density Contrast:

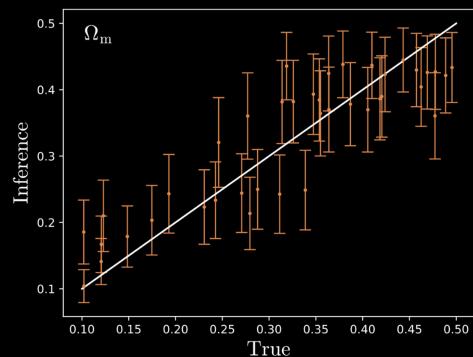
The ratio of the minimum density of the particle on the ridge of the void to the minimum density of the void

Our Work with GIGANTES using Moment Networks

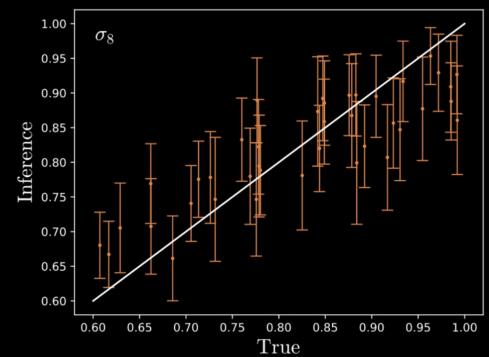
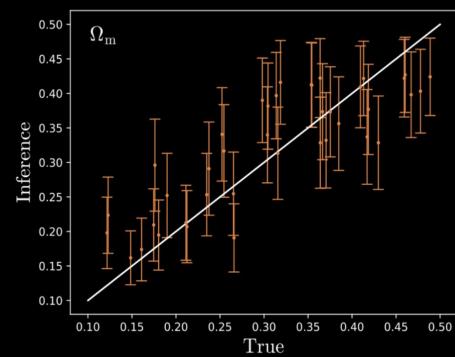


Results

Using Void Ellipticity to predict



Using Void Density Contrast to predict

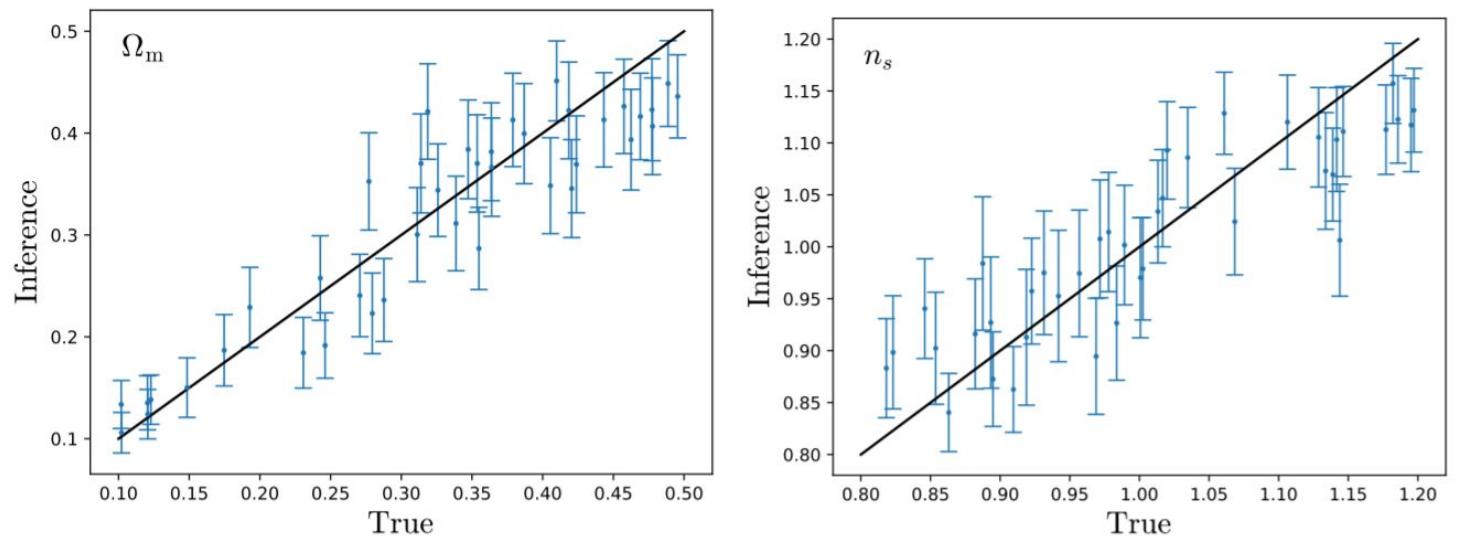


Parameter	RMSE (root-mean-square deviation)	R-Squared	χ^2_ν
Ω_m	0.0594	0.729	1.010

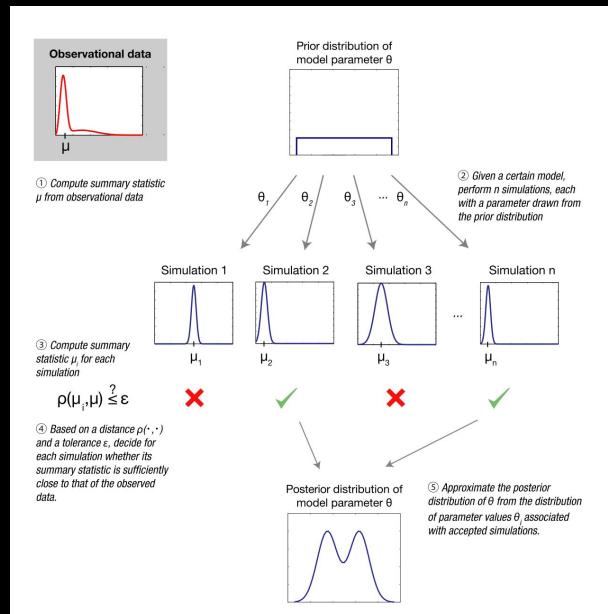
Parameter	RMSE (root-mean-square deviation)	R-Squared	χ^2_ν
Ω_m	0.0586	0.736	1.09
σ_8	0.0822	0.428	0.842

Ω_m : matter density parameter

σ_8 : The present root-mean-square matter fluctuation averaged over a sphere of radius $8h^{-1}\text{Mpc}$



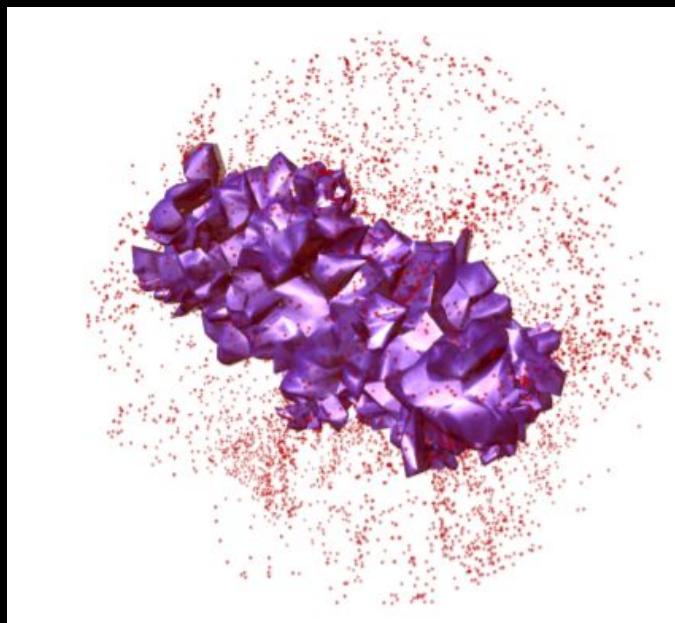
Traditional Method for Likelihood-free inference



- Approximate Bayesian computation
- Extremely time-consuming and computationally expensive (sometimes impractical) due to the high dimensionality of astrophysical data

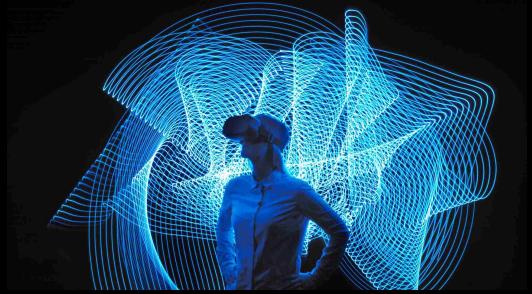
Visualize the conceptual voids –
as a stepping stone to learn more
about our mysterious universe.

How do we visualize voids? – As of NOW



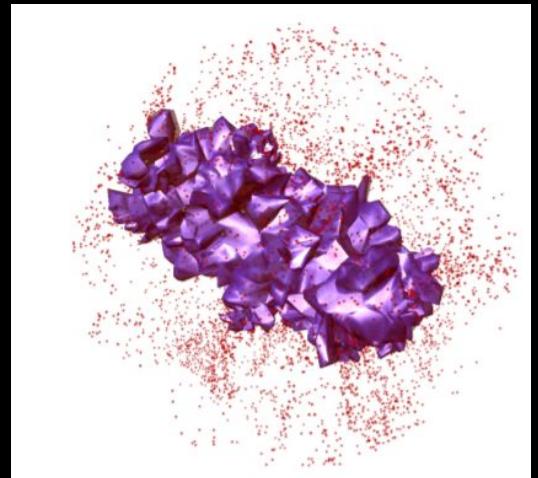
Acquaint with the Invisible

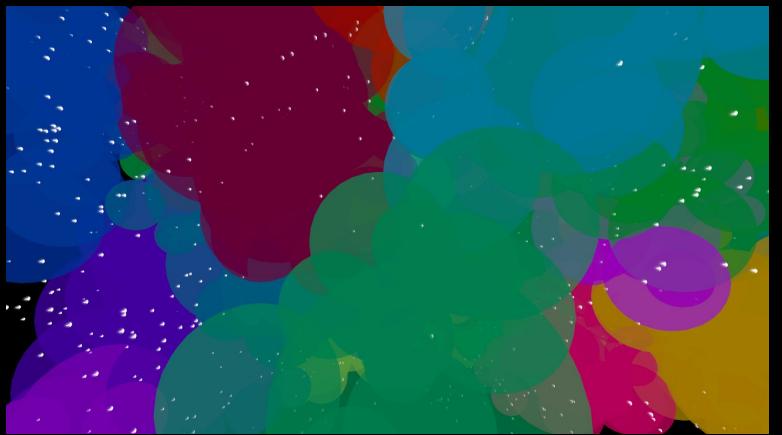
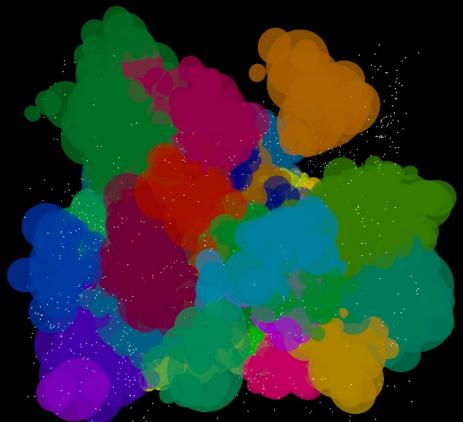
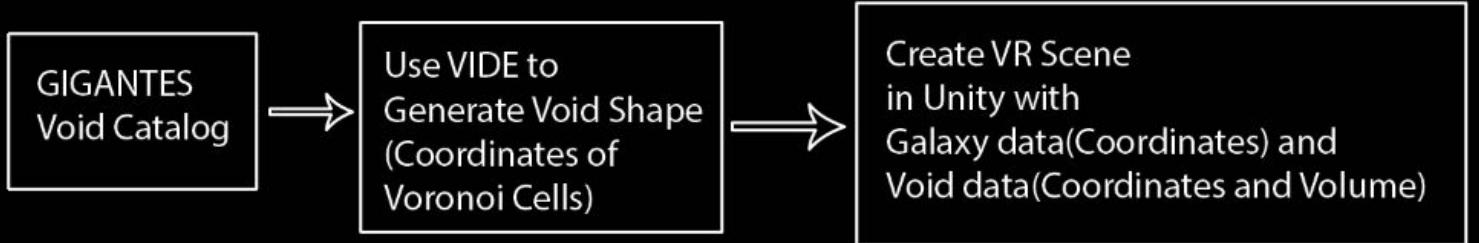
Introduce the general public to the world of invisible through VR technology – fly through and interact with cosmic voids.

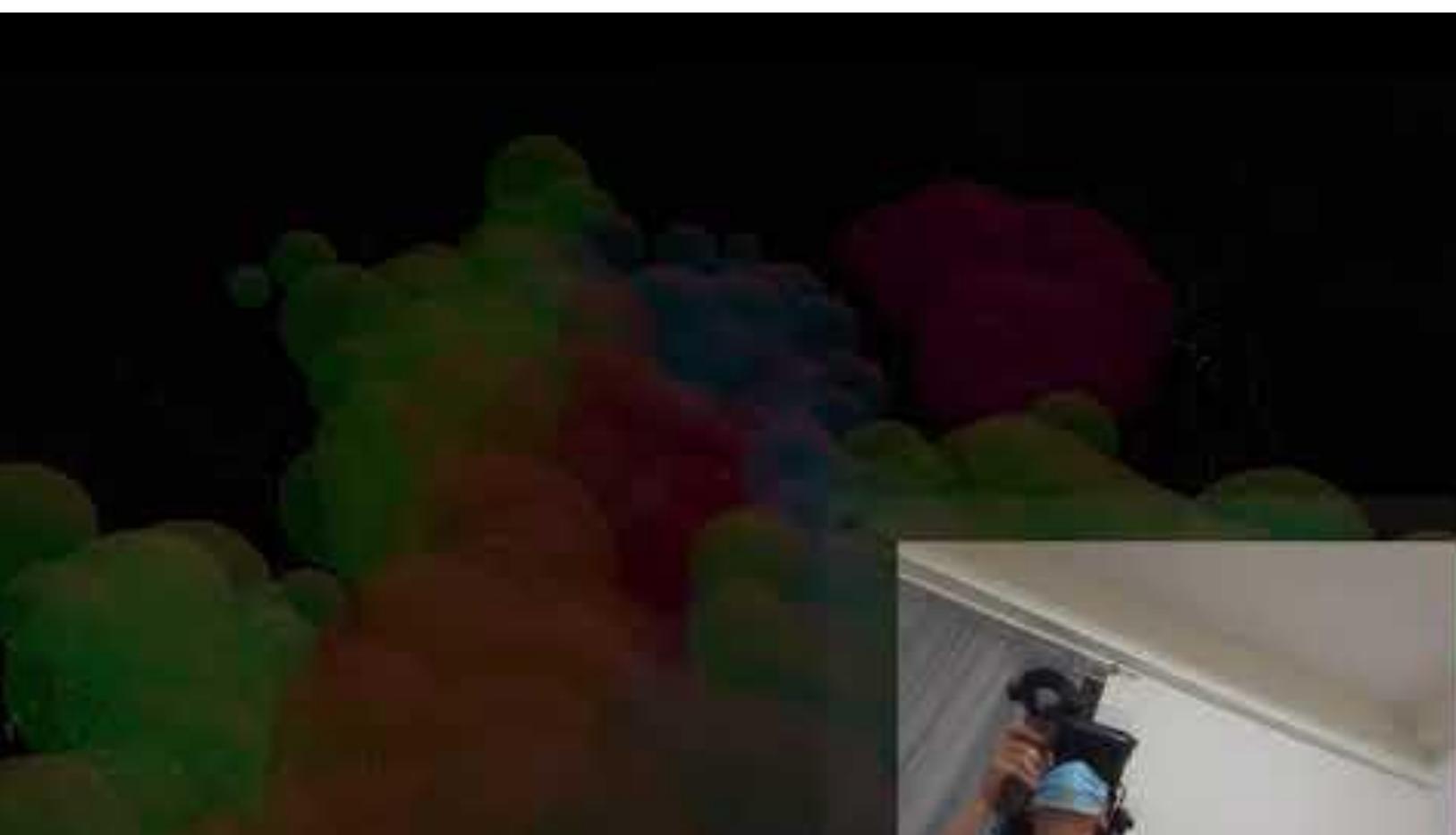


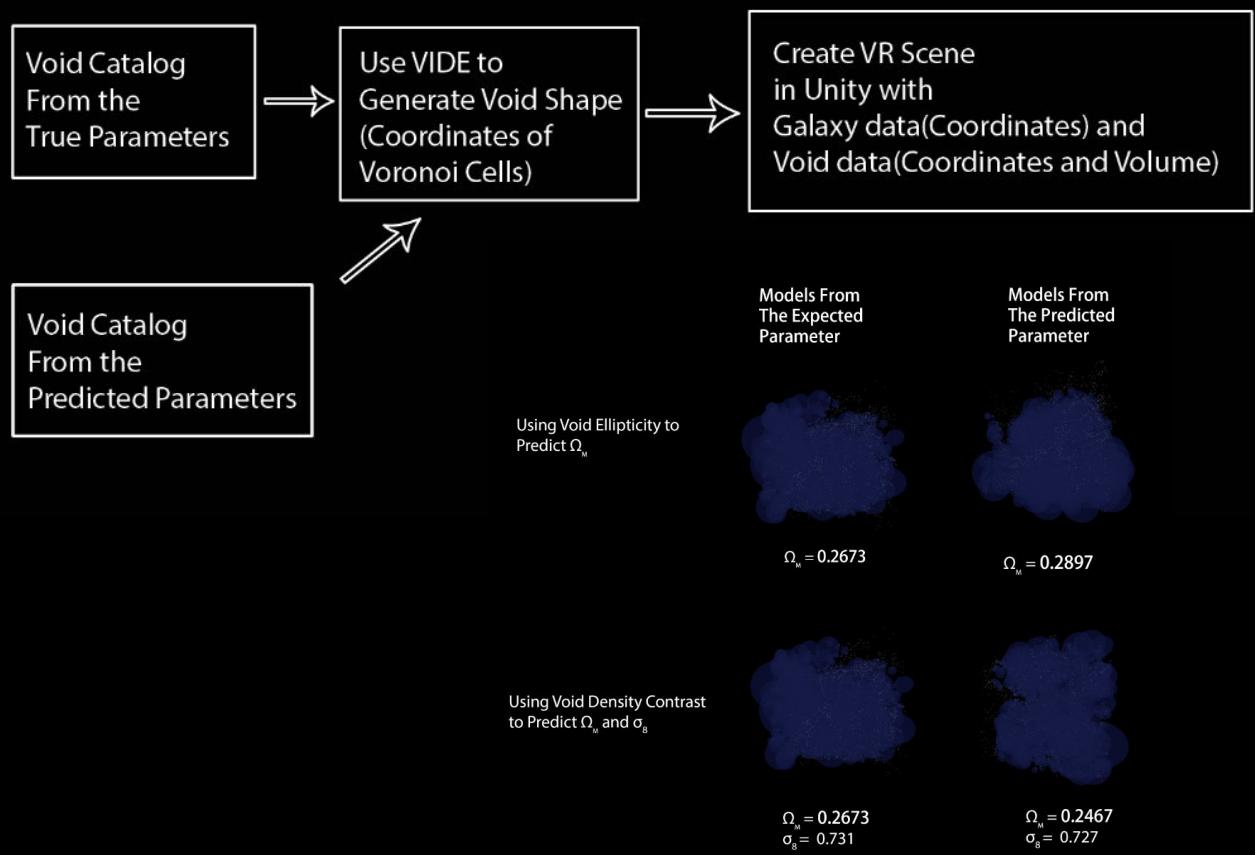
Void Visualization

- 3D VR scenes of voids
 - Better representation of the void's spatial complexity than a plain 2D graph
 - Much more immersive and interactive experience
 - Unity game engine – provides native support for Virtual Reality
 - HTC Cosmos Elite VR headset









predicted



MATLAP

Manga
Automatic
Translation
Language
Assistance
Program

Thodoris Kapouranis

Steven Lee

Motivation

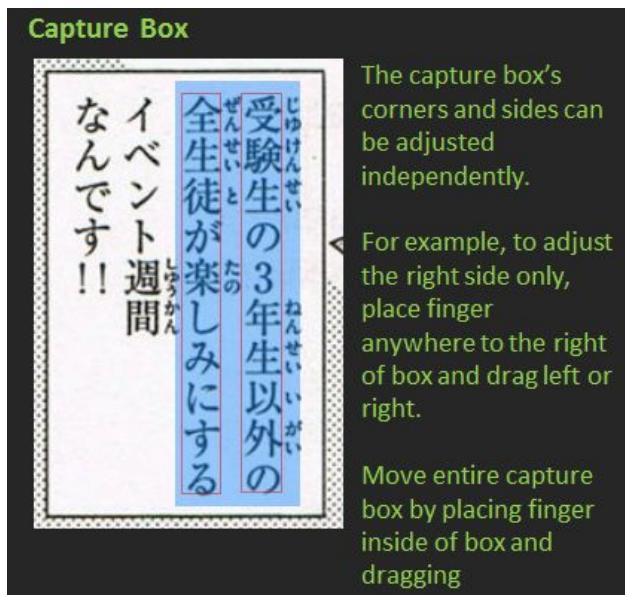
- Rise in interest in Japanese language
- Study material is saturated at the beginner level
- Large difficulty jump to start studying from native material

Objectives

- Desktop Application for mining vocabulary from comic books
- Automatic text detection
- Easy to use UI
- Less time note-taking, more time reading

Background - Typical OCR readers

- Manual, On demand OCR detection



OCR Manga Reader for Android

Background - Study tool integration

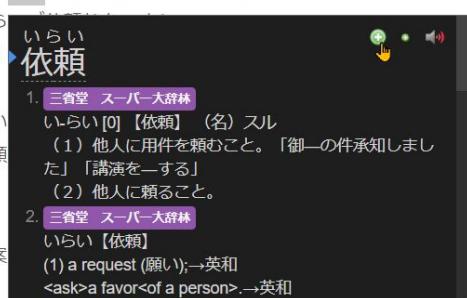
- Applications that make creating study notes easier

「あの、そちらにあるファイルも依頼票ではないのですか？」

「ああ、こちらは発明家の方から
「発明家の方だけでそんなに？」

ファイルのサイズは同じくらい
一般の依頼と発明家からの依頼
言える。

「ですが……発明家の方へのご案
「あ……」

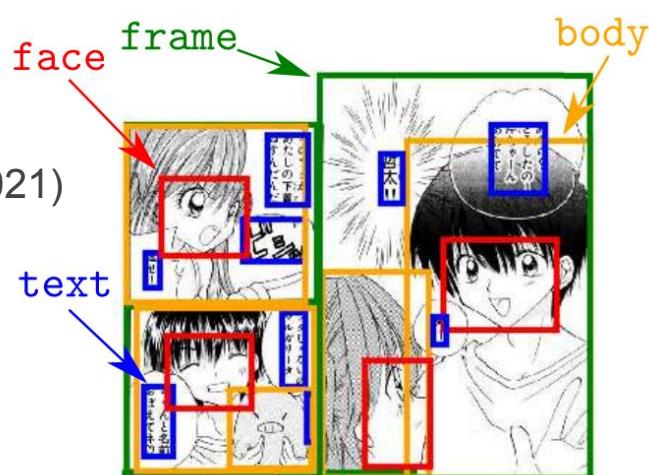


Google Chrome Extension 'Yomichan'

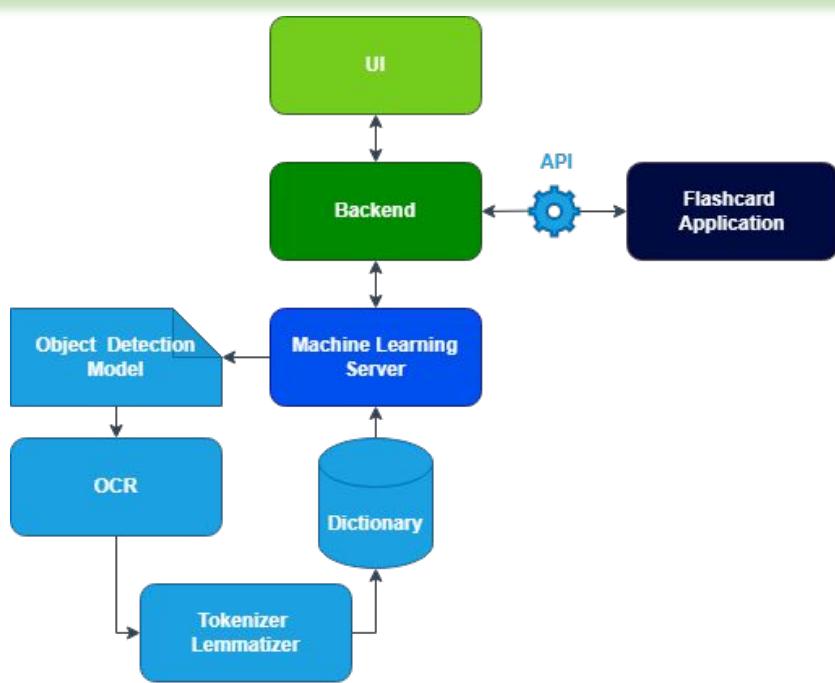
Background - Comic Object Detection

Building a Manga Dataset "Manga109" with Annotations for Multimedia Applications (2020)

"Towards Fully Automated Manga Translation" (2021)



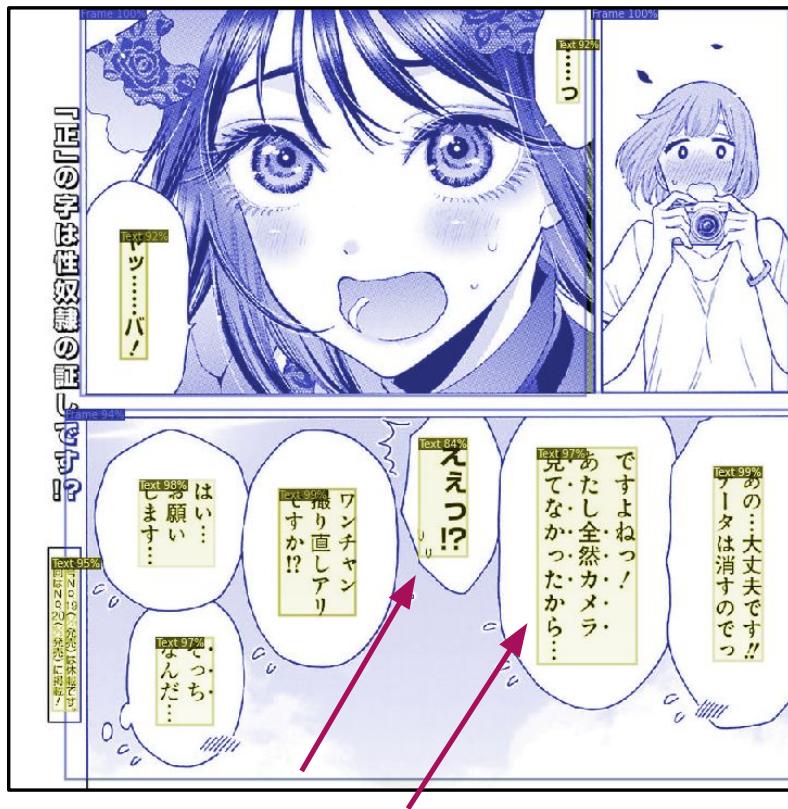
System Overview



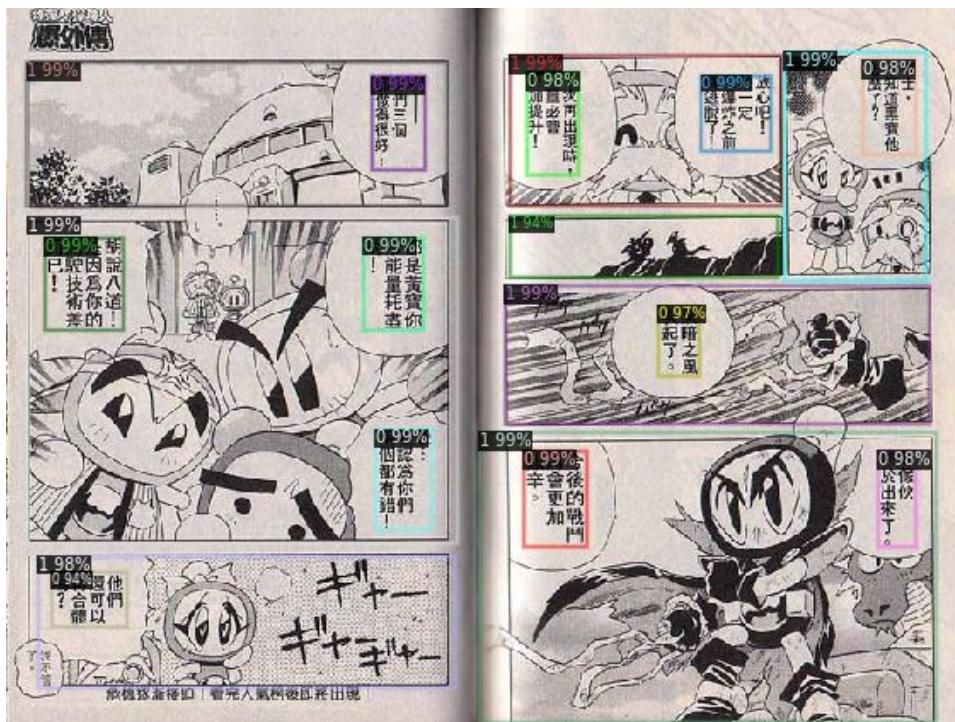


https://www.youtube.com/watch?v=xcKrOIKNEjo&feature=emb_title

Model Results - High quality scan



Model Results - Chinese language & bad scan



Real-time Personalized Sound Signal Separation & Augmentation

•••

Jungang Fang
Jinhan Zhang

Background & Problem Statement

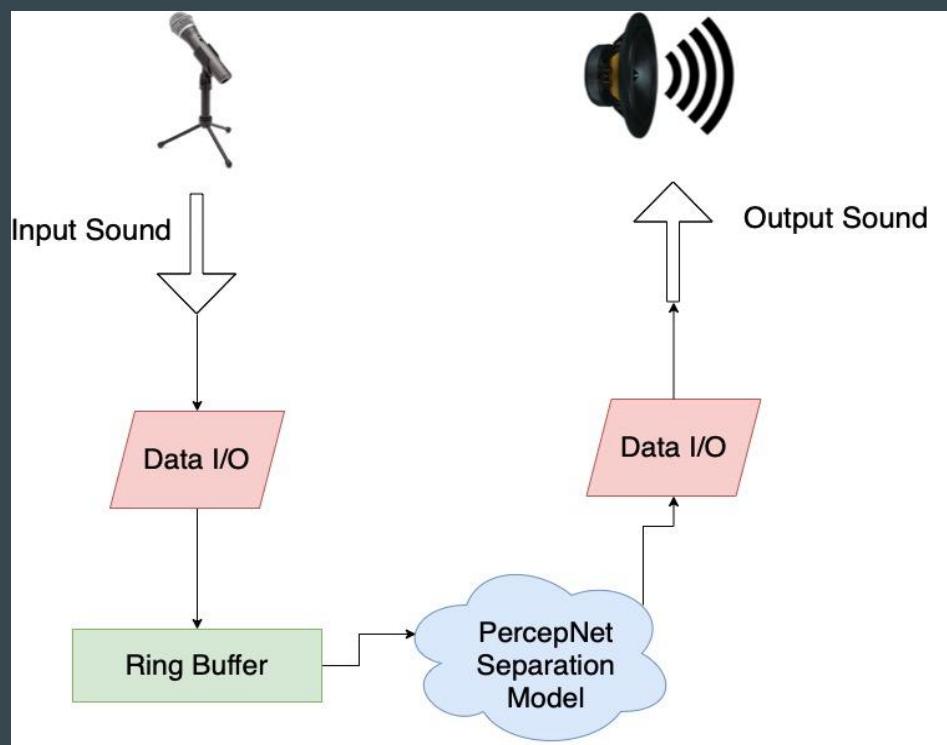
Goal:

To improve single source speech quality by denoising and speech recognition in real-time.

- ❖ bandwidth: wideband
- ❖ sample rate: 48000 Hz
- ❖ frame length: 10 ms
- ❖ trying to achieve total algorithmic latency(framesize + stride time): $\leq 60\text{ms}$

C++ Implementation of PercepNet Pipeline

- Taking real-time input from 3M worktunes headset microphone,
- process the input sound data frame by frame with the model, and generate output sound signal



Our Progress – PercepNet Implementation

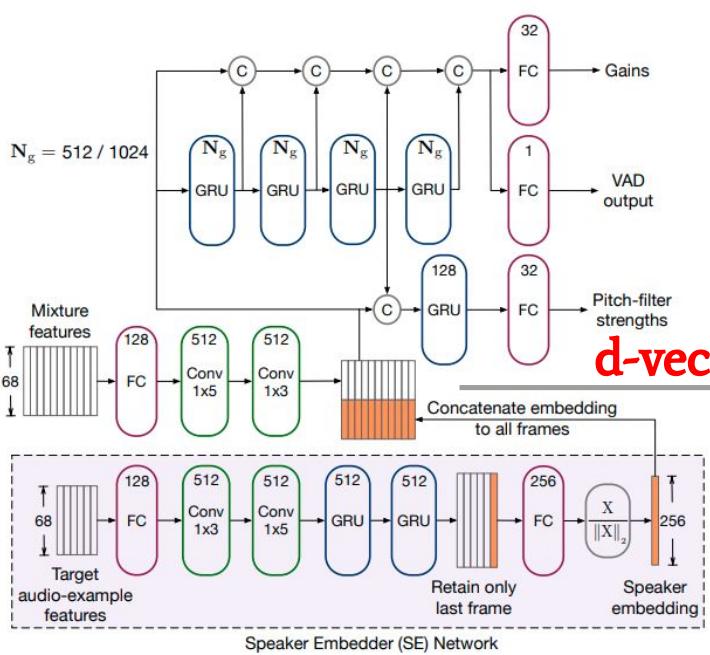
- Fully implemented PercepNet based on a partial implementation lacking the key feature ‘Envelope Postfiltering’ of the PercepNet.
- Implemented Envelope Postfiltering by modifying the gain produced by the DNN:

$$\hat{g}_b^{(w)} = \hat{g}_b \sin\left(\frac{\pi}{2} \hat{g}_b\right)$$

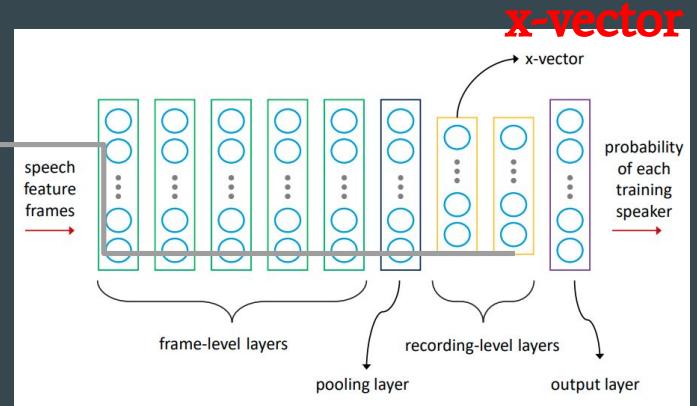
- Compensate for the global gain:

$$G = \sqrt{\frac{(1 + \beta) \frac{E_0}{E_1}}{1 + \beta \left(\frac{E_0}{E_1}\right)^2}}$$

Our Progress – Personalized Layer Deep Speaker Embedding



Deep Speaker Embedding is the extracted feature of a target speaker generated by the feature extracting model trained by the Deep Neural Network.



Our Progress – Model Training with X-vector

Personalized PercepNet: d-vector

- Cosine scoring
- Required a large number of in-domain training speakers

Our Work: x-vector

- Improved from i-vector, probabilistic linear discriminant analysis (PLDA) scoring.
- Improve performance on smaller, publicly available datasets
- instead of training the system to separate same-speaker and different speaker pairs, the DNN learns to classify training speakers

Datasets

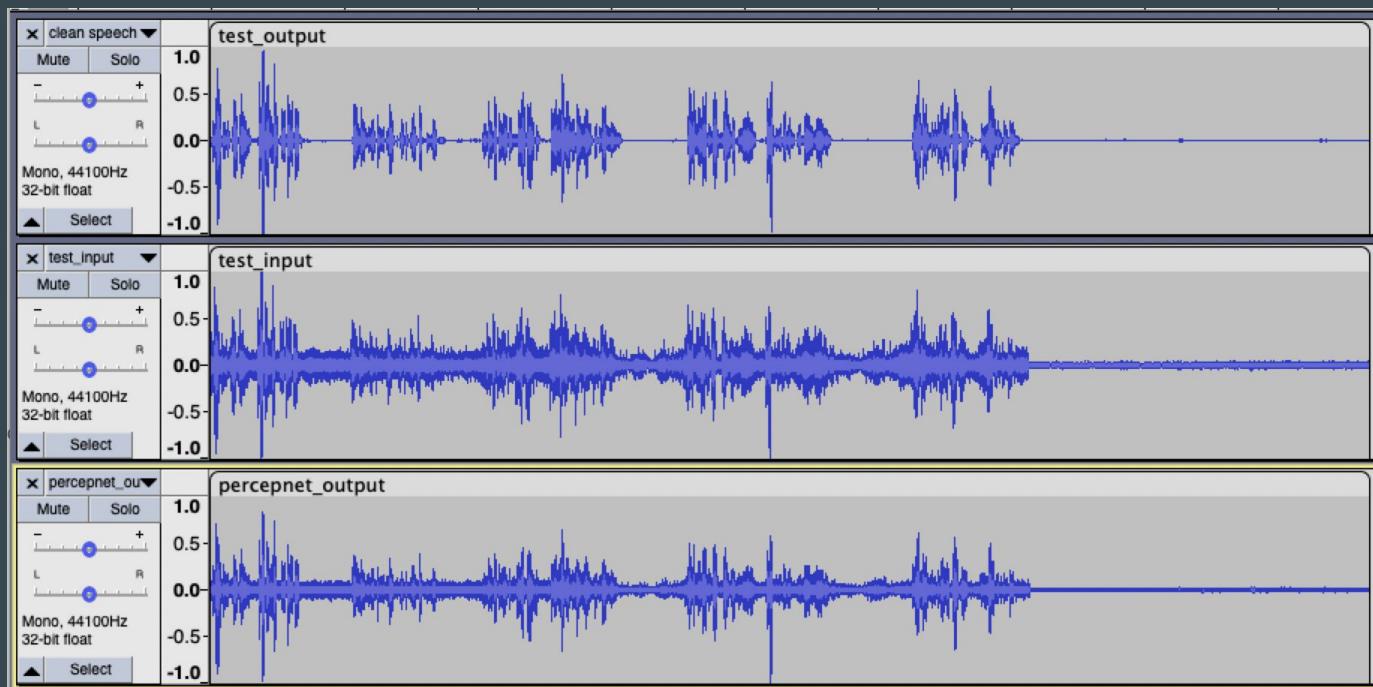
- ❖ Training PercepNet DNN
- ❖ Training Personalized Embedding DNN
 - Kaldi Speech Recognition Toolkit
- ❖ Testing PercepNet and x-vector Personalized PercepNet with PESQ

Our Progress – Model Testing with PESQ Objective Testing

- PESQ stands for Perceptual Evaluation of Speech Quality
- Objective test for scoring audio quality
- The test compares an audio output to the original voice file
- higher scores indicating better quality

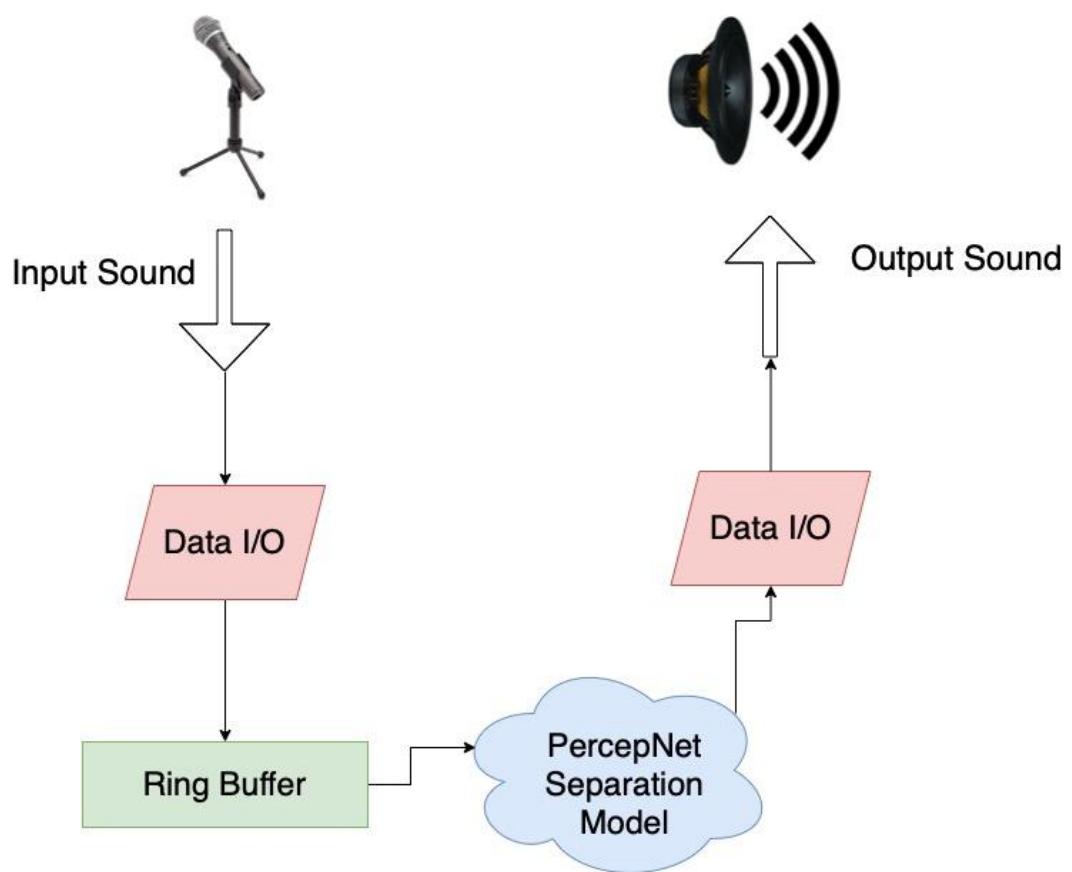
	PercepNet	x-vector Personalized PercepNet
PESQ-WB	1.55	2.36

Waveforms



Future Work

- ❖ Generate C++ implementation for the personalized layer of the model and add it to the current pipeline
- ❖ Test the model with personalized layer in real-time
- ❖ Reduce pipeline's latency



Existing Solution

- ❖ Traditional solution
 - Use multiple mics
 - The mic closer to the mouth will capture the sound energy of human voice
 - The mic further away from the mouth will capture the noise signal
 - Softwares subtract signals from each other, and generate almost clean signals
 - Difficulty
 - Multiple hardwares, hard to fit everything in a mobile device
 - Some situations cannot be handled
- ❖ DSP solution
 - Traditional DSP solutions try to find the pattern of noise signals and filter the signal frame by frame
 - Difficulty
 - Low variety, can only be used in a certain conditions, where noise signals have patterns

Separation Model

- PercepNet
 - 10ms frames
 - Ratio mask to separate signals
 - Envelope postfiltering
- Incorporate Personalized Layer to PercepNet
 - Produce a representative embedded vector
- Customize window size
 - A window size to be able to provide enough information
 - Do not cost too much latency
- Reduce the computational latency
 - Graph surgery
 - Rewire the network's components

Enhanced
Version

A Hybrid Approach for Image Vectorization for Semi-Geometric Images

by
Jonathan Lam, Derek Lee, Victor Zhang

Prof. Sam Keene

May 13, 2022

Problem statement

- ▶ Converting a raster (pixel-based) image to vector (shape-based) image
- ▶ Develop hybrid method that combines benefits of previous methods

Edge tracing

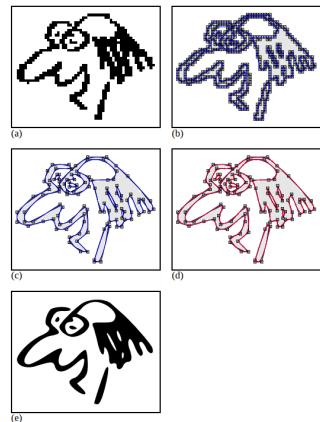


Figure: Illustration of the Potrace [1] vectorization process

Blue-noise sampling

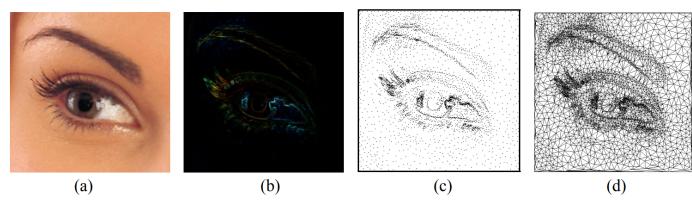


Figure: Illustration of the BNS vectorization process [2]

Hybrid approach

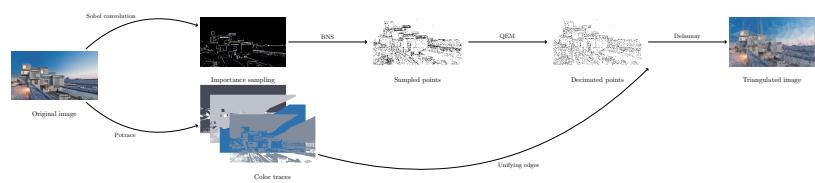


Figure: Architecture diagram

Results

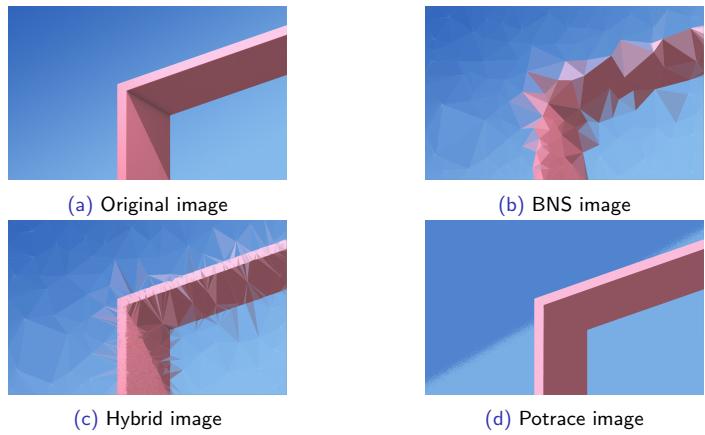


Figure: Set of images for experiment 3

Results



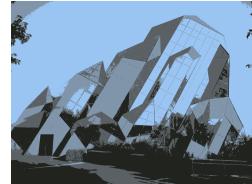
(a) Original image



(b) BNS image



(c) Hybrid image



(d) Potrace image

Figure: Set of images for experiment 4

Results



(a) Original image



(b) BNS image



(c) Hybrid image



(d) Potrace image

Figure: Set of images for experiment 5

Results

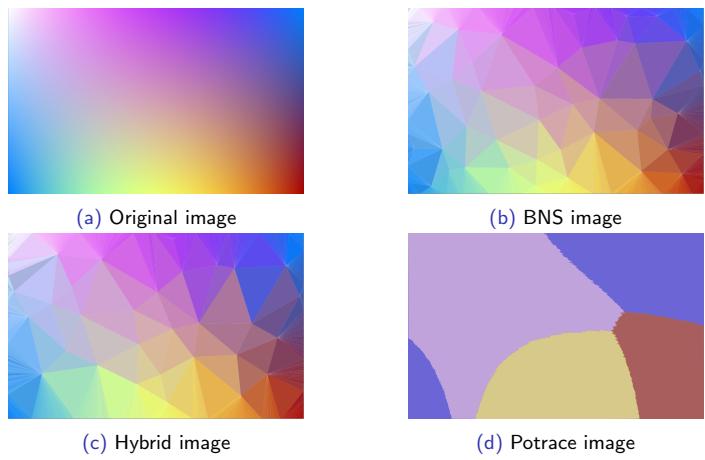


Figure: Set of images for experiment 8

Conclusions

- ▶ Implemented framework for vectorizing images
- ▶ Based on blue-noise sampling and Potrace
- ▶ Larger file size in exchange for better performance on accuracy (MSE)

Future work

- ▶ Alternative methods to strengthen edges
- ▶ Curve simplification
- ▶ Machine learning preprocessing
- ▶ Mathematical model of pipeline
- ▶ Improved evaluation metrics

References

- [1] Peter Selinger. "Potrace: a polygon-based tracing algorithm". In: *Potrace (online)*, <http://potrace.sourceforge.net/potrace.pdf> (2009-07-01) 2 (2003).
- [2] Jiaojiao Zhao, Jie Feng, and Bingfeng Zhou. "Image vectorization using blue-noise sampling". In: *Imaging and Printing in a Web 2.0 World IV*. Vol. 8664. International Society for Optics and Photonics. 2013, 86640H.

THE COOPER UNION
FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

**A HYBRID APPROACH FOR IMAGE
VECTORIZATION FOR
SEMI-GEOMETRIC IMAGES**

Jonathan Lam, Derek Lee, Victor Zhang

ECE396 Final Report
Professor Sam Keene
Spring 2022

Contents

1 Abstract	3
2 Introduction	4
2.1 Project Overview	4
2.2 Overview of Methods	5
2.3 Potential Applications	5
3 Background	6
3.1 Raster Graphics	6
3.2 Vector Graphics	6
3.3 Scalable Vector Graphics File Format	6
3.4 Content Loss	7
4 Related Work	9
4.1 Tracing Methods for Vectorization	9
4.2 Machine Learning Approaches to Vectorization	11
4.3 Sampling Methods for Vectorization	12
4.4 Vector Image Optimization	13
5 Proposed Method	14
5.1 Sampling	14
5.2 Generating Multi-Thresholded Potrace Curves	15
5.3 Improving Edges in the Mesh	18
5.3.1 Sampling Strong Edges	18
5.3.2 Unifying Edges with Potrace	18
5.4 Sampling Points Around Perimeter	19
5.5 Mesh Optimization	19
5.6 Triangle Mesh Coloring	20

5.7	Writing to SVG	20
5.8	Evaluation Metrics	21
5.9	Overview of Proposed Model	22
6	Results	24
7	Future Work	26
7.1	Experiments with the Pipeline	26
7.1.1	Order with Pipeline Components	26
7.1.2	Hyperparameter Search	26
7.2	Alternative Methods to Strengthen Edges	26
7.3	Curve Simplification	27
7.4	Image Size Sensitivity	27
7.5	Hybrid Method Runtime Performance	27
7.6	Mesh Coloring Methods	27
7.7	Using the Output of the Pipeline	28
7.7.1	Architectural Design Process	28
7.7.2	Machine Learning Preprocessing	28
7.7.3	Art Generation	28
7.8	Mathematical Model of Pipeline	28
7.9	Improved Evaluation Metrics	29
7.9.1	Controlling for Features of the Vectorized Images	29
7.9.2	Evaluation Metrics for Visual Perception	29
8	Conclusion	30
References		31
A	Appendix	34

1 Abstract

Image vectorization is the process of converting a raster (pixel-based) image to a vector (shape-based) image. While raster images are the dominant mode of image representation, vector graphics may be more efficient for highly geometric images, such as logos, fonts, and maps. Edge tracing methods for vectorization produce clean edges but assume a color-thresholded image. Sampling-based methods work well over color gradients but produce a mesh that may not be well-aligned with edges. We aim to create a hybrid pipeline that combines the benefits of these two methods: performing well over color gradients and producing clean edges. We demonstrate that our method tends to perform better in terms of accuracy (MSE) and visual presentation of edges than the base methods, at the cost of some efficiency of representation.

2 Introduction

2.1 Project Overview

Our project aims to provide an end-to-end *image vectorization* tool for a wide class of image types. Image vectorization is the process of generating a *vector* (shape-based) image that is faithful to the input *raster* (pixel-based) image.

Highly geometric images may benefit greatly from a vector image representation. For example, image vectorization for maps [3] or charts [9] have been very successful. Potrace [18] works well for shapes with well-defined geometric patches. More modern machine learning methods such as Im2Vec [17] can also identify simple geometric shapes.

We wish to study a more general class of raster images: those that remain highly-geometric (so that a vector image representation is useful), but with no other strong assumptions. For example, Potrace requires that an image is binary-thresholded or otherwise color-thresholded, and does not provide a great representation of gradient patches. The classical edge tracing methods for maps or charts also assume a color-thresholded image and may use a knowledge-based system [12] to improve the results. Im2Vec makes simplifying assumptions about the number of shapes it is attempting to identify.

A possible use case for our project is with architectural photographs. Architecture tends to be very geometric; however, complexity is introduced by many factors such as textures, coloring and lighting, fine details, and image quality, that will complicate existing methods of image vectorization. It will be useful to extract a vector representation of an architectural work from a photograph, for further use in machine learning or perhaps as a reference model in computer-aided design.

Due to the complexity of architectural images, we are not too concerned about some information loss; the goal of this project is to explore some heuristic

methods to blindly (i.e., without a knowledge-based system) produce a reasonable representation of a highly-geometric image. Several metrics will be used to quantify the performance of our vectorization method compared to the Potrace and blue-noise sampling methods.

2.2 Overview of Methods

Multiple methods have been considered for this project, including traditional edge tracing, machine learning, and blue-noise sampling (followed by triangulation). The result of our experimentation is a method that augments a sampling-based method with additional information about “strong” edges. This gives us a hybrid method that combines the benefits of sampling methods (robustness to color gradients) and edge tracing (robustness of edges).

2.3 Potential Applications

Our project can potentially be applied to the architecture design process. We envision that an architect may take a photo of an existing architectural design, use our project to process that image, and use the vector-based output to easily edit the image. Alternatively, the input image may be exported to a line-drawing representation generated from the SVG output.

Another potential use case is for machine learning. In computer vision, image data used as input is traditionally in raster format – there is little research performed on how well deep learning performs on vector-based image inputs. We imagine that due to the efficiency of its representation, especially for highly-geometric shapes, we may be able to have more concise information in the deep learning model. This representation may be used for new types of vector image-based ML models, which are currently not widely used. Alternatively, existing machine-learning methods may be used after rasterizing the output.

3 Background

3.1 Raster Graphics

Raster images are the matrix representation of an image. A raster image is conceptually a matrix of pixels, or a bitmap. Each pixel may contain multiple data points representing channels (colors). Historically, bitmaps have been the dominant representation for images due to their conceptual simplicity and the array-based display (and framebuffer) of modern screen technology. Many image-based algorithms depend on the grid-like representation of raster images, such as image compression, parallelized image processing algorithms, and image-based machine learning algorithms. As a result, standards for raster images tend to have wider support than vector graphics.

3.2 Vector Graphics

Vector images use a shape-based parameterization of an image. One of the immediate benefits is an efficient representation for purely geometric images, and the efficient and infinite scaling of geometric objects. In order to display a vector image onto a pixel-based screen, the vector image first has to be rasterized, or rendered. Vector graphics are especially useful for web graphics and other highly geometric designs, such as logos, maps, computer-aided design (CAD), and typography. However, vector-based designs tend to be more inefficient for arbitrary image data.

3.3 Scalable Vector Graphics File Format

Scalable Vector Graphics, or SVG, is a standard [16] for vector graphics that uses the XML text format. All elements are represented using combinations of seven geometric shapes: Path, Rectangle, Circle, Ellipse, Line, Polyline, and Polygon.

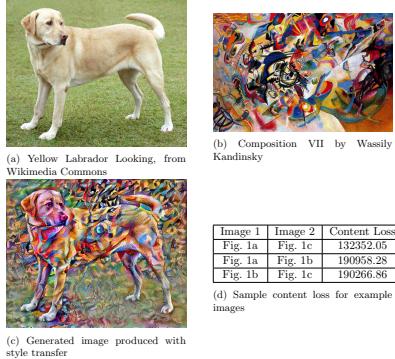


Figure 1: Content loss example

Since it is a textual format, it can be easily examined and manipulated by computers or by humans. The SVG standard is stable and supported by many modern applications, including most PDF viewers and web browsers. Due to its wide support as a vector image format, we will be using the SVG format as the default format for our vector graphics.

3.4 Content Loss

To evaluate our results, we want an evaluation metric to quantify the accuracy relative to the original image in terms of visual similarity. Content loss [5] was introduced as a loss function to train generative adversarial networks for style

transfer, and serves our purpose.

The authors define the content loss between two images as the Euclidean distance between the high-level outputs of a trained classifier. This stems from a theory about deep convolutional neural networks: the early layers in a classifier are used to extract low-level features, such as edges, while the later layers of the classifier use the low-level features to create high-level features, such as an arm or a leg.

We tested the content loss metric on three images related to style transfer¹. Fig. 1c is generated using the content from Fig. 1a and the style from Fig. 1b. The content loss between Fig. 1a and Fig. 1c should be the lowest, compared to the content loss between Fig. 1a and Fig. 1b, because the two images have the same content but with different style. As seen in Fig. 1d, we get the expected results.

¹The sample images shown come from https://www.tensorflow.org/tutorials/generative/style_transfer.

4 Related Work

4.1 Tracing Methods for Vectorization

One method of image vectorization is referred to as *tracing*. The intuition behind this is that a raster image can be thought of as a collection of adjacent image patches, and we can vectorize an image by detecting edges of shapes.

A noteworthy implementation of image tracing is the Potrace algorithm [18]. As the name suggests, Potrace first attempts to convert a raster image into a series of polygonal paths via edge detection and straight-line detection, and then attempts to simplify (optimize) polygons by reducing path cardinalities and introducing Bezier curves. It employs many useful heuristics to improve image quality, such as removing speckles smaller than a given “turd size,” detecting and smoothing corners, redundancy coding in the target format, scaling and rotating a small set of parameterized curves, and data quantization. An illustration of the stages of the Potrace algorithm is shown in Fig. 2. The implementation of Potrace is open-source, and the program is highly configurable via command-line options.

This interpretation of vectorization is useful for simple raster images that are indeed a collection of adjacent shapes, such as map data, floor charts, topography, or charts. For such images, the Potrace algorithm is both reliable and efficient. We use Potrace in our implementation to address some of the limitations in our method.

One of the drawbacks of tracing is that we can only trace edges on a binary thresholded image; if there aren’t clearly defined edges, or if there are image gradients (as is often the case), it doesn’t represent an image as well. Tracing can be applied to color images by thresholding the image by color or brightness level, and producing vector images for each thresholded layer, but this may seem choppy and low-quality. Tracing also does not recognize non-contiguous

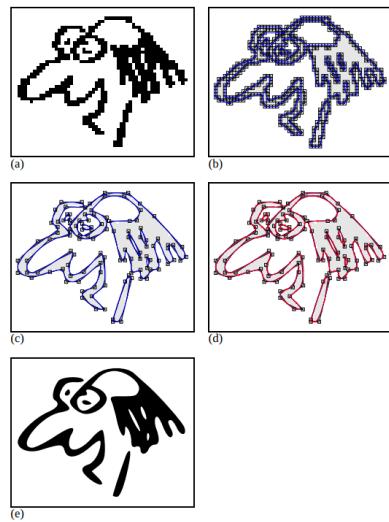


Figure 2: An illustration of the Potrace [18] vectorization process

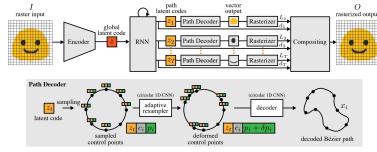


Figure 3: Architecture overview for Im2Vec from [17]

shapes (e.g., simple shapes that intersect other shapes), which can allow for more aggressive optimization or semantic segmentation (which may be achieved by specialized neural network approaches such as [11]).

Most research on tracing predates other methods, but there have been some recent innovations. For example, Yang et al. [20] implemented beziregion (bezirer region) approximation for clipart, which directly optimizes beziregions rather than going through intermediate polygons, and there have been several methods targeted at vectorizing line drawings [1, 6, 13].

4.2 Machine Learning Approaches to Vectorization

There have been several recent neural-network based approaches to image vectorization. One notable example is Im2Vec [17], a deep neural-network to vectorize images without supervision.

We explored using this model as part of our approach, but it appears that the authors hard-coded the number of shapes in the input raster image, along with the colors of each shape. We are looking to apply our model to more generic images, which will likely not be as well-defined as the emojis used in their experiments.

Neural-network-based vectorization like Im2Vec appears to be mostly limited

to simple images for the time being, but they can achieve some useful effects that may be difficult using conventional algorithms. For example, Kim et al. [11] achieves “semantic segmentation,” which is a complex task that will be difficult to approximate using classical deterministic heuristics.

4.3 Sampling Methods for Vectorization

Sampling methods tackle the vectorization problem by stochastically approximating regions of the vector image. This allows us to achieve a reasonable performance and accuracy. Sampling methods may approximate edges less accurately than edge tracing, but they can overcome some of tracing’s limitations, namely being limited to binary or multi-level thresholding. Like tracing methods, it extends fairly well to more complicated images, unlike machine learning methods, which appear to be more limited to simple images.

An example procedure for image vectorization through sampling is shown in Zhao, Feng, and Zhou [21]. The sampling method for vectorization is comprised of three steps. The image is first convolved with a Sobel differential operator to generate an *importance matrix*. Importance corresponds to spatial gradients in the original image; larger gradients may indicate regions with more detail. We then apply blue-noise sampling to the image using the importance matrix. Blue-noise sampling [19] is a general technique to non-uniformly sample an image, such that areas of higher importance are sampled at a greater density. Thus, the sampled points are more tightly clustered around more detailed regions, giving a better representation of the image. The sampled points are then triangulated using a Delaunay triangulation, which is then exported to a vector image format such as SVG.

A similar work is vectorization of cartoon images via shape subdivision [23]. This triangulates the input image, and then performs heuristics to merge trian-

gles to mitigate artifacts from triangulation. We take a similar approach, but both attempt to optimize triangles (via a quadric error metric) and augment the triangulation with additional information about “strong” edges.

4.4 Vector Image Optimization

Several methods for optimizing a polygonal path (i.e., a closed polyline) into a smaller polyline, and by expressing curved sequences of edges as a single Bezier curve, are explored in the Potrace algorithm [18]. However, this only deals with simplifying curves, while maintaining the overall topology. The sampling method generates a triangulated mesh rather than a sequence of closed paths; in this method, the optimization scheme may look different and may change the overall topology. For example, we do not have any polylines to curve-optimize unless some edges are removed to form non-triangular polygonal patches; it is then a matter of which edges can be removed while retaining fidelity to the original image.

Garland and Heckbert [7] proposed using the Quadric Error Metric (QEM) for mesh surface simplification. QEM is used for measurement of error that evaluates the distance of a point in mesh from its ideal position. The method computes the QEM for all valid pairs of points and then finds and collapses the pairs of least cost. The method works for 3D meshes and preserves the primary features of the shape. Since our algorithm first samples in 2D space based on importance, the sampled point cloud needs to be extended to 3D using color information. The resulting 3D points should be less dense but still preserve the sampling feature since decimated pairs of points in areas of high importance would result in higher loss under the QEM.

5 Proposed Method

We propose a hybrid method based on both the blue-noise sampling [21] and Potrace [18] algorithms. In this section we describe the major components of the image pipeline.

Our implementation is a mix of Python and C++². The OpenCV library [2] was used to handle image processing tasks; the `numpy` library [14] was widely used for generic numeric operations; the `open3d` library [22] was used for blue-noise sampling; and the `cairo` library [8] was used for exporting to SVG.

5.1 Sampling

The sampling process is based on Zhao, Feng, and Zhou [21]. The input image is a regular raster image, such as Fig. 4. The first step is to apply the gradient operator on the image to get the importance matrix, shown in Fig. 5. This involves convolving the image with four 3×3 Sobel filters (horizontal, vertical, and two diagonals) and finding the elementwise maximum along the four outputs to determine the magnitude of the gradient, which is interpreted as the local information content or ‘importance’ of the area surrounding a given pixel. Next, the importance matrix is thresholded so that only points of high importance are retained, shown in Fig. 6. This is essentially a high-pass filter; low-frequency components of the image are lost. The intuition is that low-frequency elements can be represented with uniformly-colored shapes without much information loss.

The code used for sampling is a C++ package provided by Ostromoukhov, Donohue, and Jodoin [15]. This method is very fast and deterministic, using Penrose tilings and Fibonacci numbers to sample using the importance map.

Now we perform the sampling to obtain Fig. 7. The sampling density at a

²The GitHub repository is <https://github.com/Victooooor/Vectorize-Arch>.



Figure 4: Original image

pixel is a function of the importance of that pixel; a higher importance leads to a higher sampling frequency. This is known as “blue-noise” sampling, and allows us to focus more detail on regions of higher information content. After sampling, the image is converted to a triangular mesh by performing the Delaunay triangulation, as shown in Fig. 8.

5.2 Generating Multi-Thresholded Potrace Curves

The Potrace command line utility does not provide the ability to perform a multiple-color thresholded scan. This multi-color scan is available in popular software such as Inkscape, but is not available as an accessible command-line tool³. As a result, we created our own pipeline for automatically generating a multi-thresholded Potrace vectorization.

Since Potrace requires binary-thresholded images, we threshold the image into its major color groups using the OpenCV implementation of a standard k -means color quantization algorithm. We then run each of the k color groups

³To the authors’ best knowledge.

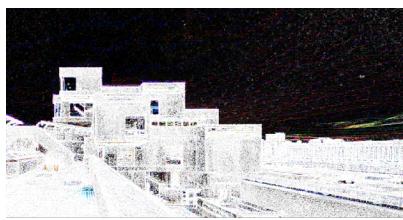


Figure 5: Importance function applied to the image

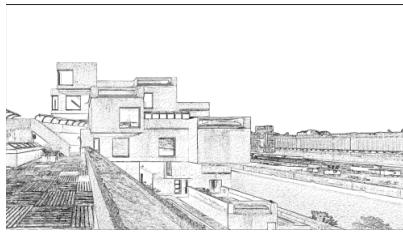


Figure 6: Thresholded points

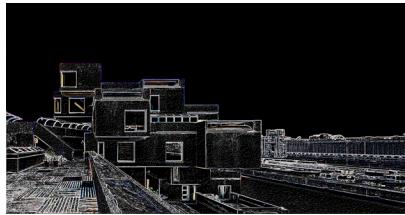


Figure 7: Sampled points

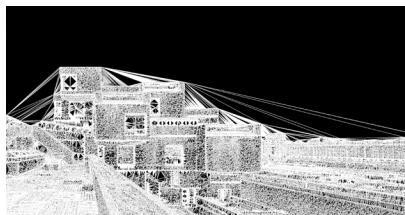


Figure 8: Triangulated image

through the Potrace algorithm separately to generate k SVG images, and merge (concatenate) the results into a single SVG.

5.3 Improving Edges in the Mesh

Blue-noise sampling attempts to represent areas of higher detail with a higher sampling density. The greatest issue with the blue-noise sampling is that it does not represent edges well. Ideally, many of the edges in the image will align with edges of the mesh. We can attempt to manually adjust the sampled points to lie along edges. The following two heuristics are used to try to improve edges.

5.3.1 Sampling Strong Edges

We may attempt to augment edges by simply increasing the number of samples along the edge. To do this, we identify the set of points that form “strong edges.” This may come from an edge detection algorithm; in our case, we choose points which have an importance (gradient) above an arbitrary threshold. Then we randomly sample from this set with some small probability, and use these sampled points to augment those from blue-noise sampling. We note that this adds points and increases the file size. This increase in points is proportional to the number of pixels that lie on “strong edges.”

5.3.2 Unifying Edges with Potrace

Another method to improve edges is to move sampled points that lie close to an edge to that edge. We use a custom heuristic to combine our sampled points with the Bézier curves that are generated by Potrace. We first rasterize the Bézier curves in order to be able to compare the curves with our sampled points. Next, we replace our sampled points with the closest point on a Bézier curve if the closest point is within a certain distance of the sampled point.

If there are multiple points on the Bézier curve within the same distance, we use all of the points. This distance is determined by the attraction distance ϵ , which represents a square centered on a point on the Bézier curve. The square's corners are ϵ pixels up/down and left/right of that point. For example, one corner would be at $(+\epsilon, -\epsilon)$ relative to the point on the curve. If the sampled point is within this square, the point on the Bézier curve is a valid candidate for replacement. We check all of the Bézier curves generated by Potrace, and for each sampled point, we replace it with the closest point across all of the Bézier curves, assuming a closest point exists.

For our implementation of this algorithm, we first preprocess the sampled points in order to efficiently find all valid sampled points that are within the given distance (determined by ϵ). We accomplish this by creating a 2D matrix that is the same size as our image, where $cell_{i,j}$ represents all sampled points within the given distance. We iterate through each sampled point, and insert that sampled point into all indices within a square centered at that sampled point.

5.4 Sampling Points Around Perimeter

In order for the triangulated mesh to cover the whole image, we need to add additional points around the perimeter of the image before triangulation. The current implementation uniformly samples points around the perimeter. While this does not guarantee that every pixel is part of the mesh, it causes most of the pixels to be covered.

5.5 Mesh Optimization

To increase the efficiency of the representation, we wish to reduce the number of points from the sampling without greatly reducing accuracy. To do this,

we decimate points following the method by Garland and Heckbert [7], which decimates points in such a way to optimize the quadric error metric (QEM). To transform the 2D sampled point cloud to 3D, we use the color information at the sampled points as the third dimension. The resulting 3D point cloud is then simplified using the QEM decimation method implemented by the `open3d` library [22].

5.6 Triangle Mesh Coloring

We experiment with three methods for determining the color of a mesh triangle. The first method takes the mean of the RGB values at the three points of the triangle. The second method randomly samples a set number of integer points inside of the triangle and takes the mean of the RGB values of those points. The third method takes the mean of the RGB values at every integer point in the triangle. We note that the third method approximates the triangle, so it may sample integer points outside of the triangle. In addition, the third method utilizes the first method for small triangles, where the number of integer points inside the triangle is less than three. We currently use the second method, which randomly samples points inside of the triangle. This method has good performance and gives good color estimates for the triangles.

5.7 Writing to SVG

The `pycairo` and `cairosvg` libraries are used to export the triangulated representation to SVG and PNG files. The PNG file is a rendered (rasterized) version of the SVG image, used for content loss and MSE metrics.

5.8 Evaluation Metrics

We evaluate based on several metrics. A couple of our metrics are familiar: file size and mean-square error (MSE). We also implement a custom content loss metric based on Dumoulin, Shlens, and Kudlur [5]. Similar to the methodology used by the authors, we use a pre-trained VGG-19 and strip off the *conv₅* and fully-connected blocks. We note that although the authors used a VGG-16, we use a VGG-19, similar to Huang and Belongie [10]. In order to get the content loss between two images, we take the Euclidean distance between the outputs of the stripped VGG-19.

5.9 Overview of Proposed Model

Name	Default Value	Description
Importance Scalar	100.0	A positive scalar used at the stage of importance sampling, the importance matrix is scaled by this parameter. A higher value results in higher sampling density.
Decimation Scalar	10	A constant larger than 1, used to calculate the expected number of points for QEM mesh simplification. The expected number of points is calculated as the number of sampled points divided by the decimation scalar. A higher value results in stronger point decimation.
Potrace Scans	4	A positive integer, the number of scans of different color threshold chosen by k-means clustering. Same as number of colors in the resulting SVG.
Attraction Distance	15	A positive integer, the maximum distance (L^∞) between sampled points and curve generated by Potrace for the point to tangency point.
Edge Density	0.01	A float in the range (0,1), the probability of a point on an edge to be chosen. A higher value results in a higher point density for an edge.

Table 1: List of hyper-parameters

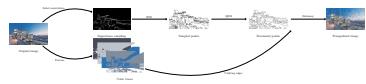


Figure 9: Architecture diagram

6 Results

As seen by Table 2, Potrace produces images that are almost always smaller than our hybrid method. Merging the blue-noise sampling points with Potrace usually significantly increases the file size, although in rare cases, such as with simple images (e.g. Fig. 17), this can reduce the file size.

As seen by Table 3, Potrace almost always performs better than our hybrid method, according to the content loss metric. However, it is important to note that content loss may not be the best metric for measuring the accuracy of the vectorized image relative to the original raster image. For example, as seen by Fig. 17, the Potrace image does not look similar to the original image at all, yet its content loss is lower than both the BNS method and our hybrid method.

As seen by Table 4, our hybrid method consistently performs the best (with the exception of a single experiment), according to the MSE metric. Our hybrid method likely performs better than the BNS method because our hybrid method typically has significantly more points. Additionally, our hybrid method likely performs better than Potrace because it is able to capture color more effectively. Potrace is restricted to a predetermined number of unique colors (we use four colors in our experiments), while our method can use as many unique colors as possible.

On a more qualitative measure, Fig. 13 shows that our hybrid method significantly improves the edges in our final method. The BNS method captures the colors, but the edges are unrecognizable.

Image	BNS	Hybrid	Potrace
1	367	485	267
2	203	196	106
3	126	405	399
4	208	2890	1596
5	696	4270	5879
6	358	1369	702
7	299	5723	1725
8	89	77	23
9	205	327	70

Table 2: File size (kB) by method

Image	BNS	Hybrid	Potrace
1	110	108	84
2	84	84	62
3	75	68	42
4	141	98	89
5	125	89	90
6	128	105	86
7	116	66	47
8	67	71	60
9	81	71	99

Table 3: Content loss ($\times 10^3$) by method

Image	BNS	Hybrid	Potrace
1	63.02	61.75	88.85
2	45.73	40.87	67.10
3	96.89	10.11	79.03
4	75.35	47.76	92.25
5	86.29	77.17	88.86
6	69.59	62.58	87.65
7	59.63	37.03	77.63
8	50.86	56.15	100.04
9	79.21	60.41	100.76

Table 4: MSE (rounded to nearest hundredth) by method

7 Future Work

7.1 Experiments with the Pipeline

7.1.1 Order with Pipeline Components

There is a large design space for our the model. While the plain BNS method is relatively simple (perform importance map, sample, and triangulate), our method involves many more components intended to improve the performance of edges, and decimates points to achieve some improvement in representational efficiency. Transposing or moving certain components of the pipeline (e.g., moving the QEM decimation step to after merging with Potrace) may result in improved results. This requires a large number of additional experiments.

7.1.2 Hyperparameter Search

In addition to moving large components around, performing an exhaustive hyperparameter grid search may be useful to determine which parameters generate the best results.

7.2 Alternative Methods to Strengthen Edges

Currently, two heuristic methods are used to strengthen edges. There is a large design space for both of these methods.

The current algorithm to merge points with Potrace, as described in Section 5.3.2, is fairly simple. It is a quadratic ($\mathcal{O}(N^2)$) algorithm, and thus fairly slow. Also, it uses a simple L^∞ distance metric rather than a Euclidean distance, out of simplicity. There may be a better algorithm from computational geometry to perform the merging task.

Points are randomly sampled from a set of “strong edge points,” which are points for which the importance (gradient) is above a certain threshold.

Alternatively, points may be sampled from edges generated from another edge detection method, such as the Canny edge detection algorithm [4].

7.3 Curve Simplification

The result of the proposed algorithm is a triangulated mesh. While this is a simple representation and easily generated using the Delaunay triangulation algorithm, it is not especially visually appealing, especially when groups of edges may be joined together into Bezier curves.

It may be beneficial to perform curve optimizations to improve visual output similar to Selinger [18]. Yang et al. [20] works directly with beziregions, although this may be difficult to integrate into our pipeline.

7.4 Image Size Sensitivity

Certain aspects of our pipeline are sensitive to file size, such as the importance scaling factor, which affects the density of sampling. It may be beneficial to ensure that all parameters are image-size-invariant.

7.5 Hybrid Method Runtime Performance

Runtime or algorithmic performance was not a major concern for this work, as it was a secondary goal to improving the quality of vectorization. However, it is a very practical concern, and should be considered in the future. Some of the pipeline may be amenable to parallelization, such as the sampling and Potrace multi-scan passes.

7.6 Mesh Coloring Methods

We briefly experimented with three methods for coloring triangles in the mesh. However, further experiments would be useful in determining which method is

most beneficial (also taking into account runtime performance).

7.7 Using the Output of the Pipeline

7.7.1 Architectural Design Process

The original intention for our model was to aid in the architectural design process. It will be useful for architects to generate line drawings or CAD models. Our model does not currently do this, but such a representation may benefit from the mesh outputted by our model.

7.7.2 Machine Learning Preprocessing

Most, if not all, image machine learning models take raster images as input. In order to use our proposed method as input for a machine learning model, the output must be rasterized, or a vector-based machine learning model must be developed. This lies outside the scope of our work.

7.7.3 Art Generation

The outputted vectorized images have a distinct “stained glass” texture that may be desirable in an art context. A number of peers have said that the output “looks cool” and this may be desirable simply for visual effect.

7.8 Mathematical Model of Pipeline

The nature of this work is highly heuristic and empirical; it would be pleasing to have a mathematical model behind the enhancements to estimate the gains in vectorization and to check the empirical results against.

7.9 Improved Evaluation Metrics

7.9.1 Controlling for Features of the Vectorized Images

The current evaluation metrics are very basic. Moreover, when comparing images in an experiment, we do not attempt to control features of the output images (e.g., number of mesh points, filesize, etc.) but instead only control the model parameters. In other words, the current comparison may be less fair than if the outputted images of BNS and our hybrid method had the same filesize.

7.9.2 Evaluation Metrics for Visual Perception

Much of the motivation behind this work is highly related to visual perception: Potrace appears to do worse for gradients, and BNS tends to produce much worse edges. However, none of the evaluation metrics directly test either of these visual-based perceptions.

We use content loss as an attempt to measure overall semantic fidelity of the vectorized image, but this may not be the best metric. For example, in experiment 8, which contains the gradient, the Potrace output has a lower content loss despite being much less accurate than the other methods. We hypothesize that a better metric may be style loss [5].

8 Conclusion

We have implemented a basic framework for vectorizing raster images, primarily based on blue-noise sampling [21] and Potrace [18]. Our proposed framework involves performing blue-noise sampling on an image, merging sampled points with the Potrace output, triangulation, and exporting to an SVG file. While this method currently does not generate a highly efficient representation, it performs better than BNS and Potrace on some metrics, particularly accuracy. Our method is able to both handle gradient patches better than Potrace, and represent edges better than blue-noise sampling. There is much future work remaining to further tune this model for efficiency.

References

- [1] Mikhail Bessmeltsev and Justin Solomon. "Vectorization of line drawings via polyvector fields". In: *ACM Transactions on Graphics (TOG)* 38.1 (2019), pp. 1–12.
- [2] Gary Bradski and Adrian Kaehler. "OpenCV". In: *Dr. Dobb's journal of software tools* 3 (2000), p. 2.
- [3] Girija Dharmaraj. *Algorithms for automatic vectorization of scanned maps*. Citeseer, 2005.
- [4] Lijun Ding and Ardesir Goshtasby. "On the Canny edge detector". In: *Pattern recognition* 34.3 (2001), pp. 721–725.
- [5] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. *A Learned Representation For Artistic Style*. 2017. arXiv: 1610.07629 [cs.CV].
- [6] Jean-Dominique Favreau, Florent Lafarge, and Adrien Bousseau. "Fidelity vs. simplicity: a global approach to line drawing vectorization". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–10.
- [7] Michael Garland and Paul Heckbert. "Surface Simplification Using Quadric Error Metrics". In: *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics* 1997 (July 1997). doi: 10.1145/258734.258849.
- [8] Cairo Graphics. *Cairo: A 2D graphics library with support for multiple output devices, version 1.1*. 2006.
- [9] Weihua Huang, Chew Lim Tan, and Wee Kheng Leow. "Elliptic arc vectorization for 3D pie chart recognition". In: *2004 International Conference on Image Processing, 2004. ICIP'04*. Vol. 5. IEEE, 2004, pp. 2889–2892.
- [10] Xun Huang and Serge Belongie. *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*. 2017. arXiv: 1703.06868 [cs.CV].

- [11] Byungsoo Kim et al. "Semantic segmentation for line drawing vectorization using neural networks". In: *Computer Graphics Forum*. Vol. 37, 2. Wiley Online Library. 2018, pp. 329–338.
- [12] Kyong-Ho Lee, Sung-Bae Cho, and Yoon-Chul Choy. "Automated vectorization of cartographic maps by a knowledge-based system". In: *Engineering Applications of Artificial Intelligence* 13.2 (2000), pp. 165–178.
- [13] Gioachino Noris et al. "Topology-driven vectorization of clean line drawings". In: *ACM Transactions on Graphics (TOG)* 32.1 (2013), pp. 1–11.
- [14] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.
- [15] Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. "Fast Hierarchical Importance Sampling with Blue Noise Properties". In: *ACM Transactions on Graphics* 23.3 (2004). Proc. SIGGRAPH 2004, pp. 488–495. url: <http://www.iro.umontreal.ca/~ostrom/ImportanceSampling/>.
- [16] Antoine Quint. "Scalable vector graphics". In: *IEEE MultiMedia* 10.3 (2003), pp. 99–102.
- [17] Pradyumna Reddy et al. *Im2Vec: Synthesizing Vector Graphics without Vector Supervision*. 2021. arXiv: 2102.02798 [cs.CV].
- [18] Peter Selinger. "Potrace: a polygon-based tracing algorithm". In: *Potrace (online)*, <http://potrace.sourceforge.net/potrace.pdf> (2009-07-01) 2 (2003).
- [19] Dong-Ming Yan et al. "A survey of blue-noise sampling and its applications". In: *Journal of Computer Science and Technology* 30.3 (2015), pp. 439–452.
- [20] Ming Yang et al. "Effective clipart image vectorization through direct optimization of bezigons". In: *IEEE Transactions on Visualization and Computer Graphics* 22.2 (2015), pp. 1063–1075.

- [21] Jiaojiao Zhao, Jie Feng, and Bingfeng Zhou. “Image vectorization using blue-noise sampling”. In: *Imaging and Printing in a Web 2.0 World IV*. Vol. 8664. International Society for Optics and Photonics. 2013, 86640H.
- [22] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. “Open3D: A modern library for 3D data processing”. In: *arXiv preprint arXiv:1801.09847* (2018).
- [23] Ju Jia Zou and Hong Yan. “Cartoon image vectorization based on shape subdivision”. In: *Proceedings. Computer Graphics International 2001*. IEEE. 2001, pp. 225–231.

A Appendix

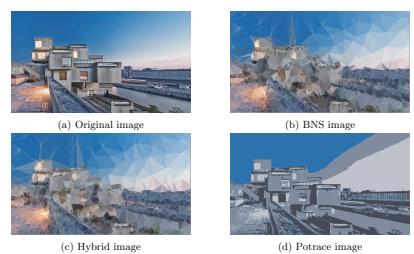


Figure 10: Set of images for experiment 1

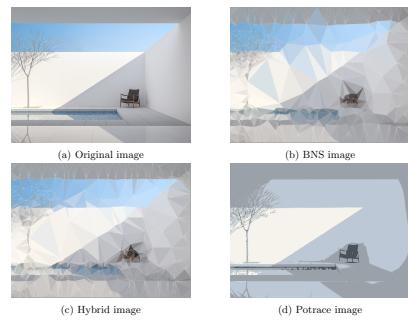


Figure 11: Set of images for experiment 2

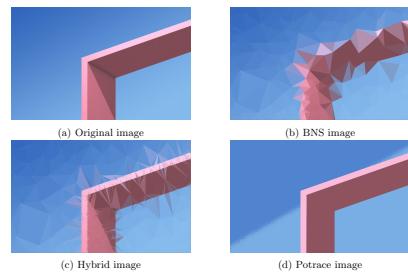


Figure 12: Set of images for experiment 3

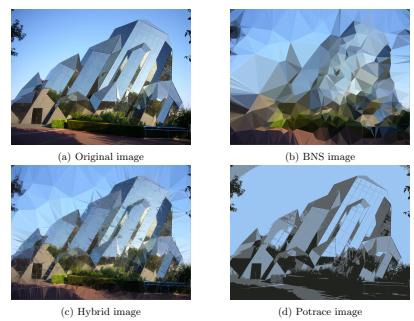


Figure 13: Set of images for experiment 4



Figure 14: Set of images for experiment 5

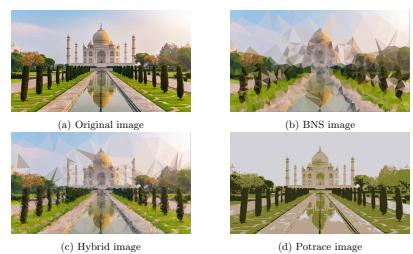


Figure 15: Set of images for experiment 6

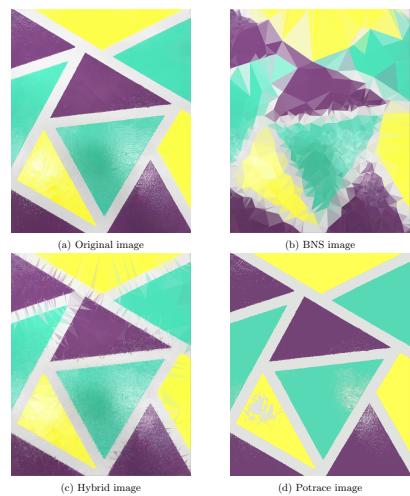


Figure 16: Set of images for experiment 7

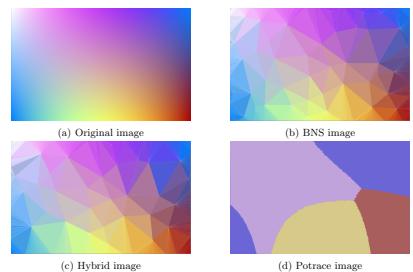


Figure 17: Set of images for experiment 8

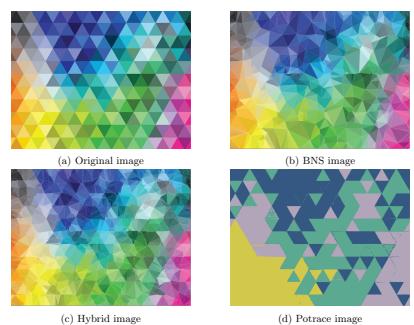


Figure 18: Set of images for experiment 9

THE COOPER UNION
FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

**IMAGE VECTORIZATION
FOR ARCHITECTURE**

Jonathan Lam, Derek Lee, Victor Zhang

ECE395 Mid-year Report
Professor Sam Keene
December 2021

Contents

1 Abstract	3
2 Introduction	4
2.1 Goal	4
2.2 Previous Works and Motivation	4
2.3 Methods	4
2.4 Potential Applications	5
3 Background	6
3.1 Raster Graphics	6
3.2 Vector Graphics	6
3.3 Scalable Vector Graphics File Format	7
3.4 Content Loss	7
4 Related Work	11
4.1 Tracing Methods for Vectorization	11
4.2 Machine Learning Approaches to Vectorization	13
4.3 Sampling Methods for Vectorization	14
4.4 Vector Image Optimization	15
5 Methods	17
5.1 General	17
5.2 Sampling	17
5.3 Mesh Optimization	20
5.4 Triangle Mesh Coloring	20
5.5 Writing to SVG	21
5.6 Evaluation Metric	21
5.7 Implementation stack	23

6 Conclusion	24
7 Future Work	25
8 References	26

1 Abstract

Architecture projects often involve both large structures and detailed components, and are often highly geometric. Digital architecture designs (i.e., created through the use of CAD) are often stored in a vector (shape-based) format for convenient editing. Raster (pixel-based) images, such as photographs, are not as easily used for these purposes, which largely diminishes their usability for the design process. Our project aims to develop a image vectorization method (i.e., a tool to convert from raster to vector format) specialized towards architecture images and explores the potential of vector-based images in the architecture design process and in machine learning preprocessing.

There are several general methods for image vectorization – we propose a sampling-based vectorization method involving three steps. The image is sampled using a blue-noise sampling technique, which extracts the high frequency components in the image and filters out the less important pixels. The sampled point cloud is then simplified to reduce the number of vectors in the final drawing. Finally, the last step involves converting the point cloud to an efficient vector representation. This vector representation is saved as some vector image format, such as SVG.

2 Introduction

2.1 Goal

Our project aims to provide an end-to-end image vectorization tool. Vectorization is the process of generating a vector image that is faithful to the input raster image. We aim to specialize in architectural images specifically, which should allow us to optimize our project for this use case. Architectural images are usually highly geometric, which should allow for an efficient vector representation. In theory, a highly geometric image should have a vector representation that is more efficient than the original image. Of course, with real images, the image will not be perfectly geometric. However, we are comfortable with information loss, as long as the main content of the image is preserved; we will develop a suitable loss metric to quantify the representation efficiency and error loss.

2.2 Previous Works and Motivation

Some of the first image vectorization tools implemented edge tracing, which works well for simple shapes, and especially in black-and-white (or similarly color-thresholded) images. However, architectural images are more complex and can have wide range of colors, rendering traditional methods ineffective. A new vectorization method is needed to work effectively for colored complex images and to generate vector representations that are easy to use for architects.

2.3 Methods

Multiple methods have been considered for this project, including traditional edge tracing, machine learning, and sampling. These methods are compared, and we determined that the sampling method is the most effective for our use case.

2.4 Potential Applications

Our project can potentially be applied to the architecture design process, along with vector-based machine learning. One of the use cases we envision is to take a photo of an architectural design, use our project to process that image, and use the vector-based output to easily edit the image. Another potential use case is for vector-based machine learning. In computer vision, image data used as input is traditionally in raster format – there is little research performed on how well deep learning performs on vector-based image inputs. We imagine that due to the efficiency of its representation, especially for highly-geometric shapes, we may be able to have more useful information in the deep learning model from the outset. In other words, a conversion to vector-based models, in which the shapes contain meaningful information about the image, may be useful as a machine learning preprocessing step.

3 Background

3.1 Raster Graphics

Raster images are the simple matrix representation of an image. A raster image is conceptually a matrix of pixels, or a bitmap. Each pixel may contain multiple data points representing channels (colors). Historically, bitmaps have been the dominant representation for images due to their conceptual simplicity and the array-based display (and framebuffer) of modern screen technology. Much effort has been spent in enhancing the compression (e.g., lossy JPEG compression vs. lossless PNG compression) and analysis of raster images. Many image analysis methods depend on the grid-like representation of raster images, such as in the case of efficient parallel computation. Standardized raster images tend to have better support than vector graphics on older devices.

3.2 Vector Graphics

Vector graphics are not represented as pixels, but rather as a collection of parameterized geometric shapes. One of the immediate benefits is an efficient representation for purely geometric images, and the efficient and perfect scaling of continuous geometric objects. In order to display a vector image onto a pixel-based screen, the vector image first has to be rasterized, or rendered. Vector graphics are especially useful for web graphics and other highly geometric designs, such as maps, CAD, and typography. However, vector-based designs tend to be more inefficient for arbitrary image data (with high entropy), due to the relative complexity of shapes to pixels.

3.3 Scalable Vector Graphics File Format

Scalable Vector Graphics, or SVG, is a standard for vector graphics that uses the XML text format. All elements are represented using combinations of seven geometric shapes: Path, Rectangle, Circle, Ellipse, Line, Polyline, and Polygon. Since it is a textual format, it can be easily examined and manipulated by computers or by humans. The SVG standard is stable and supported by many applications, including PDF viewers and web browsers.

3.4 Content Loss

Content loss was introduced in Dumoulin, Shlens, and Kudlur [1]. The purpose of content loss is to quantify the difference in content between two images. Content loss was introduced as a loss function to train generative adversarial networks for style transfer.

The authors define the content loss between two images as the Euclidean distance between the high-level outputs of a trained classifier. This stems from a theory about deep convolutional neural networks: the early layers in a classifier are used to extract low-level features, such as edges, while the later layers of the classifier use the low-level features to create high-level features, such as an arm or a leg.

We tested the content loss metric on three images related to style transfer. The images were from https://www.tensorflow.org/tutorials/generative/style_transfer.



Figure 1: Yellow Labrador Looking, from Wikimedia Commons



Figure 2: Composition VII by Wassily Kandinsky



Figure 3: Generated image produced with style transfer

Fig. 3 is generated using the content from Fig. 1 and the style from Fig. 2. The content loss between Fig. 1 and Fig. 3 should be the lowest, compared to the content loss between Fig. 1 and Fig. 2, because the two images have the same content but with different style.

4 Related Work

4.1 Tracing Methods for Vectorization

One of the successful earlier methods in image vectorization is called tracing, and at the time was synonymous with image vectorization. The general intuition behind this is that a raster image can be thought of as a collection of adjacent image patches, and we can vectorize an image by detecting edges of shapes.

A noteworthy implementation of image tracing is the Potrace algorithm [5]. As the name suggests, Potrace first attempts to convert a raster image into a series of polygonal paths via edge detection and straight-line detection, and then attempts to simplify (optimize) polygons by reducing path cardinalities and introducing Bezier curves. It employs many useful heuristics to improve image quality, such as removing speckles smaller than a given “turd size”, detecting and smoothing corners, redundancy coding in the target format, scaling and rotating a small set of parameterized curves, and data quantization. An illustration of the stages of the Potrace algorithm is shown in Fig. 4. The implementation of Potrace is open-source, and the program is highly configurable via command-line options.

This interpretation of vectorization is useful for simple raster images that are indeed a collection of adjacent shapes, such as map data, floor charts, topography, or charts. For such images, the Potrace algorithm is both reliable and efficient. While we do not use Potrace in our implementation, we may borrow some of its features related to curve optimization when simplifying the shapes (this is future work).

One of the drawbacks of tracing is that we can only trace edges on a binary thresholded image; if there aren’t clearly defined edges, or if there are image gradients (as is often the case), it doesn’t represent an image as well. Tracing can be applied to color images by thresholding the image by color or brightness

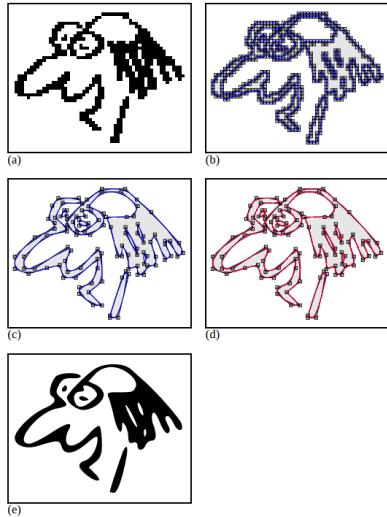


Figure 4: An illustration of the Potrace [5] vectorization process

level, and producing vector images for each thresholded layer, but this may seem choppy and low-quality. Tracing also does not recognize non-contiguous shapes (e.g., simple shapes that intersect other shapes), which can allow for more aggressive optimization and better object recognition. Machine learning approaches are better at recognizing this (citation).

4.2 Machine Learning Approaches to Vectorization

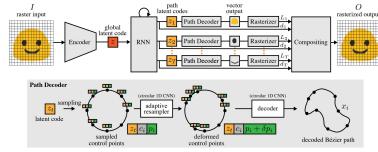


Figure 5: Architecture overview for Im2Vec from [4]

Im2Vec is an end-to-end deep neural network introduced by Reddy et al. [4]. The model takes a raster image as input and outputs an SVG image. The raster image is first passed through an encoder. The encoded representation is then passed through a Recurrent Neural Network (RNN), specifically a bidirectional Long Short-Term Memory network (LSTM), to produce an arbitrary number of outputs. Each output is passed through a path decoder, which produces a Bézier path.

The path decoder uses continuous deformation of the unit circle to ensure each path is closed. The authors sample points on the unit circle and concatenate the output of the RNN to each of the sampled points. The authors next use 1D convolutions, equivalent to convolution around the perimeter of the circle,

to determine how to deform the unit circle.

This model is trained by appending a differentiable rasterizer to the output. The rasterizer converts the vector output back to raster format and the resulting image can be compared to the original image to compute a loss. Since the rasterizer is differentiable, that loss can be backpropagated through the model.

We explored using this model as part of our approach, but after looking through the codebase, we discovered that this would likely not work for our purposes. It appears that the authors hard-coded the number of shapes in the input raster image, along with the colors that each shape should be. We are looking to apply our model to various types of architectures, which will likely not be as well-defined as the emojis used in their experiments.

4.3 Sampling Methods for Vectorization

Sampling methods tackle the vectorization problem by stochastically approximating regions of the vector image. This allows us to achieve a reasonable performance and accuracy. Sampling methods may approximate edges less accurately than edge tracing, but they can overcome some of tracing’s limitations, namely being limited to binary thresholding. Like tracing methods, it extends fairly well to more complicated images, while machine learning methods currently appear to be more limited to simple images.

An example procedure for image vectorization through sampling is shown in Zhao et al.’s work Zhao, Feng, and Zhou [6]. The sampling method for vectorization is composed of 3 steps. The image is first convoluted with a Sobel differential operator to generate an “importance matrix”. This represents the discontinuities or gradients in the original matrix; larger gradients may indicate regions with more detail. We then apply blue-noise sampling to the image using the importance matrix to determine sampling density around each pixel.

with a larger gradient causing a higher sampling density. The sampled points are then triangulated, and the line segments of the triangles form the vector representation of the image.

4.4 Vector Image Optimization

Several methods for optimizing a polygonal path (i.e., a closed polyline) into a smaller polyline, and by expressing curved sequences of edges as a single Bezier curve, are explored in the Potrace algorithm [5]. However, this only deals with simplifying curves, while maintaining the overall topology. The sampling method generates a triangulated mesh rather than a sequence of closed paths; in this method, the optimization scheme may look different and may change the overall topology. For example, we do not have any polylines to curve-optimize unless some edges are removed to form non-triangular polygonal patches; it is then a matter of which edges can be removed while retaining fidelity to the original image.

Hoppe [2] describes an approach to reduce triangular meshes by merging two adjacent vertices in an arbitrary n -dimensional triangular mesh. This technique also results in a triangular mesh, and thus may be useful for 3-D modeling. The method attempts to optimize volume preservation using a quadric error metric and color attribute discontinuities across triangle bounds, illustrated in Fig. 6.

We can be more aggressive than Hoppe's method and not preserve triangular regions, since we are working in a two-dimensional space. We can use their idea of respecting color discontinuities to remove edges, and then perform Potrace's curve optimizations on the resulting polygonal areas.

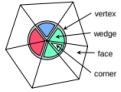


Figure 4: Wedge-based mesh representation.

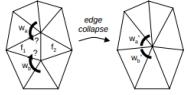


Figure 5: Tests for wedge unification after edge collapse.

Figure 6: Collapsing an edge in [2], taking into consideration color discontinuities

5 Methods

5.1 General

The end goal of the project is to develop an efficient system for image vectorization. The system would take any image as input and output an SVG representation of the original image. The system is mainly composed of 3 different components.

5.2 Sampling

The sampling process is based on [6]. The input image is a regular bitmapped image, such as Fig. 7. The first step is to apply the gradient operator on the image to get the importance matrix, shown in Fig. 8. This involves pixel-wise convolving with four 3×3 Sobel filters (horizontal, vertical, and two diagonals) and finding the magnitude to determine the magnitude of the gradient, which is interpreted as the local information content or “importance” of the area surrounding a given pixel. Next, the importance matrix is thresholded so that only points of high importance are retained, shown in Fig. 9. This is essentially a high-pass filter; low-frequency components of the image are lost. The intuition is that low-frequency elements can be represented with uniformly-colored shapes without much information loss.

Now we perform the sampling to obtain Fig. 10. The sampling density at a pixel is a function of the importance of that pixel; a higher importance leads to a higher sampling frequency. This is known as “blue-noise” sampling, and allows us to focus more detail on regions of higher information content. After sampling, the image is converted to a triangular mesh by performing the Delaunay triangulation, as shown in Fig. 11.



Figure 7: Original image



Figure 8: Importance function applied to the image

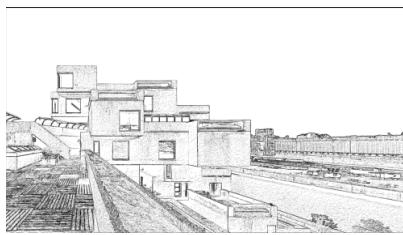


Figure 9: Thresholded points

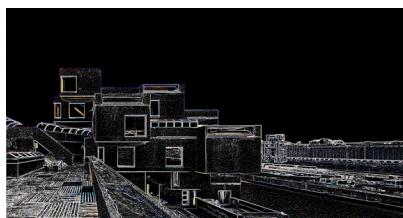


Figure 10: Sampled points

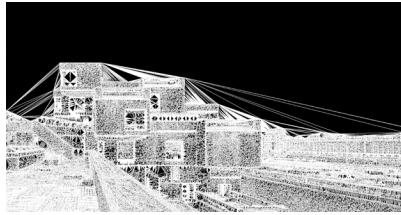


Figure 11: Triangulated image

5.3 Mesh Optimization

This is a future goal for spring semester and has not been implemented yet.

5.4 Triangle Mesh Coloring

We currently have three methods for determining what color to use for each triangle. The first method takes the mean of the RGB values at the three points of the triangle. The second method randomly samples a set number of integer points inside of the triangle and takes the mean of the RGB values of those points. The third method takes the mean of the RGB values at every integer point in the triangle. We note that the third method approximates the triangle, so it may sample integer points outside of the triangle. In addition, the third method utilizes the first method for small triangles, where the number of integer points inside the triangle is less than three.

5.5 Writing to SVG

We convert internal representations of triangles (tuples with 3 pairs, representing the coordinates of each point on the triangle) to an SVG file. We use Python's `cairo` library to accomplish this.

In the future, we may also attempt to optimize this, such as by coding redundant information.

5.6 Evaluation Metric

We implemented an evaluation metric based on content loss from Dumoulin, Shlens, and Kudlur [1]. Similar to the methodology used by the authors, We use a pre-trained VGG-19 and strip off the `conv5` and fully-connected blocks. It is important to note that the authors used a VGG-16. We use a VGG-19, similar to Huang and Belongie [3]. In order to get the content loss between two images, we take the Euclidean distance between the outputs of the stripped VGG-19.



Figure 12: Original image



Figure 13: Vectorized image created with Potrace

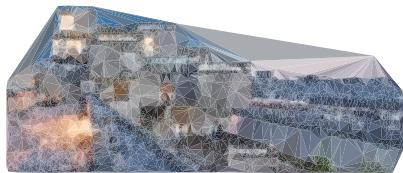


Figure 14: Vectorized image created with our method

Method	Content Loss
Our Method	118758.836
Potrace	141575.78

Table 1: Content loss for vectorizing a sample image with different methods

As seen in Table 1, our method produces a significantly lower content loss

than Potrace.

5.7 Implementation stack

The current implementation is a mix of Python and C++. The image processing components utilize OpenCV for GPU support and will be multi-threaded. Efficiency of the process is not a concern at this point; we care more about optimizing for the error metric. Future iterations of this project may consider efficiency of implementation.

6 Conclusion

Image vectorization is the process of converting a raster image to a vector image, and may lead to efficiency gains for highly vectorized images. While vectorization has been used successfully on simple vector-based input images in the past such as map or typography images, we aim to improve its output on highly-geometric but less-exact images, such as architectural images. We believe that this may be useful in the architectural design process, and perhaps in other design processes whose subject is highly geometric.

We have implemented a basic framework for vectorizing raster images, primarily based on [6]. We can take an input image, perform sampling on the image, triangulate the sampled points, and produce an output image. So far we have yet to optimize this method towards architecture – this will involve optimization stages after producing the mesh.

7 Future Work

In the upcoming semester, we would like to experiment with different vectorization approaches. This may involve different sampling methods or different optimizations. We may choose a different method for generating vectors from the sampled points. Currently, we are using a standard Delaunay triangulation, although there may be alternative methods available.

We may change our evaluation metric if we discover a better metric. We will rank each of our approaches according to our evaluation metric. By the end of the upcoming semester, we will also produce a presentation describing our results, along with a final report that contains the final decisions made for our project.

An approximate timeline for the spring semester may look like:

January Begin work on mesh optimization, review prior work.

February Continue mesh optimization, clearly define evaluation metrics.

March Begin gathering results and evaluating using given metric.

April Finish result-gathering, begin final report and presentation.

May Complete final report and presentation, present results.

8 References

- [1] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. *A Learned Representation For Artistic Style*. 2017. arXiv: [1610.07629](https://arxiv.org/abs/1610.07629) [cs.CV].
- [2] Hugues Hoppe. “New quadric metric for simplifying meshes with appearance attributes”. In: *Proceedings Visualization'99 (Cat. No. 99CB37067)*. IEEE. 1999, pp. 59–510.
- [3] Xun Huang and Serge Belongie. *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization*. 2017. arXiv: [1703.06868](https://arxiv.org/abs/1703.06868) [cs.CV].
- [4] Pradyumna Reddy et al. *Im2Vec: Synthesizing Vector Graphics without Vector Supervision*. 2021. arXiv: [2102.02798](https://arxiv.org/abs/2102.02798) [cs.CV].
- [5] Peter Selinger. “Potrace: a polygon-based tracing algorithm”. In: *Potrace (online)*, <http://potrace.sourceforge.net/potrace.pdf> (2009-07-01) 2 (2003).
- [6] Jiaojiao Zhao, Jie Feng, and Bingfeng Zhou. “Image vectorization using blue-noise sampling”. In: *Imaging and Printing in a Web 2.0 World IV*. Vol. 8664. International Society for Optics and Photonics. 2013, 86640H.

ECE491 Advanced compilers final report:
Implementing a lazy functional language

Jonathan Lam

2022/05/12

Contents

1 Project overview	3
1.1 Motivation and overview	3
1.2 Commentary on the tutorial	3
1.3 Implementation setup	4
2 Background	5
2.1 Definition of a programming language	5
2.2 Implementation(s) of a programming language	5
2.3 The untyped λ -calculus	6
2.4 Functional programming	7
2.5 Terminology	8
3 The Core language	11
3.1 Syntax	11
3.2 Dynamic semantics	11
3.3 Base definitions	13
3.4 Sample programs	14
3.5 Lexer	14
3.6 Parser	14
3.7 Pretty-print utility	15
4 Template instantiation (TI) evaluator	16
4.1 Intuitive walkthrough of a TI evaluation	16
4.2 The TI abstract machine	19
4.2.1 Unwinding	19
4.2.2 Instantiation	20
4.2.3 Advanced instantiations	20
4.2.4 Memory model	21
4.2.5 Updates and indirections for call-by-need evaluation	21

5 G-Machine (GM) implementation	22
5.1 Sample compilation	22
5.2 List of opcodes	28
5.3 A simpler stack addressing mode	29
5.4 Compilation schemes	30
5.4.1 Non-strict vs. strict compilation context	31
5.5 Evaluator	32
5.5.1 Transpilation to x86-64 assembly code	33
6 Future work	34
6.1 Garbage collection	34
6.2 Three-address machine and parallel G-machine	34
6.3 Particulars of the STG and Haskell's implementation	34
6.4 Compilation to native binaries	34
7 Conclusions	35
8 References	35
A Code samples	36
A.1 Core standard library	36
A.2 Project Euler 1	38
A.3 <i>letrec</i> demonstration	38
A.4 Infinite streams	38
A.5 Twice twice twice	39

	Purely functional	Non-purely functional
Untyped	Core	Scheme ¹
Typed	Hazel ²	OCaml ³

Table 1: Summary of programming language implementation projects

1 Project overview

1.1 Motivation and overview

This project was the semester-long project for the ECE491 independent study on advanced compilers. It is the implementation of a simple lazy (normal-order) evaluator for functional languages similar to Haskell. The work follows the tutorial, “Implementing functional languages: a tutorial” by Simon Peyton Jones at Microsoft Research [5]. This was published two years after Haskell 1.0 was defined in 1990 [2], to which SPJ is a major contributor.

This project is the culmination of my studies in programming languages and functional programming over the past two years. These studies include writing a C compiler in C (Spring 2021), a Scheme (LISP) interpreter in Scheme (Summer 2021), and this project, a Core interpreter and compiler in Haskell (Spring 2022). Additionally, my Master’s thesis (2021–2022 school year) was about updating the evaluation model of Hazel, a programming language implemented in OCaml. We summarize these languages in Table 1.

1.2 Commentary on the tutorial

The tutorial used for this project assumes a basic understanding of functional programming and a non-strict language such as Miranda or Haskell. For this report, we do not assume this and give a brief introduction to these ideas.

The text is largely in tutorial format, in that it provides much of the code for the reader to follow along with, but it is also in large part similar to a textbook. While much of the base code is given, much of the implementation is left in the form of non-trivial exercises to be completed using the provided theory. As the book progresses, less code is provided directly; rather, the semantics are given using the state transition notation, and the reader is asked to implement this in code.

The structure of the tutorial goes as follows: Chapter 1 introduces the Core language. Chapter 2 provides an evaluator implementation called the Template Instantiation (TI) evaluator, which we will describe in Section 4. Chapter 3 provides a compiled version of the TI evaluator, called the G-Machine⁴ (GM), which we will describe in Section 5. We note that Haskell officially uses a Spinless Tagless G-Machine (STG) [4].

¹Mostly functional

²Gradually-typed

³Weakly-typed

⁴“G” for “graph,” most likely.

Chapter 4 describes the Three-Address Machine compiled implementation, and Chapter 5 describes the Parallel G-Machine. The G-Machine (a stack-based machine) may be translated into a TAM representation. Chapter 6 introduces the λ -abstraction syntax using the λ -lifting program transformation. Chapters 4-6 were not covered for this independent study.

Rather than developing each implementation vertically (e.g., finishing the lexer, then the parser, then the intermediate representation, then exporting opcodes), the tutorial uses a horizontally incremental development method. In this style of development, we first complete a base implementation of only the core language features, and then incrementally add support for new language features. The tutorial calls these horizontal versions “marks,” i.e., “T1 Mark 3” or “GM Mark 7.”

I greatly enjoyed the tutorial and found the exercises delightfully challenging and enlightening. My only complaints with the tutorial are that sometimes the exercises seemed to require knowledge from future marks, and that the names of certain concepts seem to be arbitrarily named⁵.

1.3 Implementation setup

My implementation is called `f1c`, short for “fun(ctional) lazy compiler.” `f1c` is written in Haskell, similar to the tutorial⁶. Instructions for use may be found on the GitHub’s README.

The implementation may be found at [jlam5555/fun-lazy-compiler](https://github.com/jlam5555/fun-lazy-compiler). A transpiler for the GM Mark 1 to x86-64 assembly code may be found at [jlam5555/f1c-transpiler](https://github.com/jlam5555/f1c-transpiler).

⁵For example, the compilation schemes are named with arbitrary letters, and the “G” in “G-Machine” and the “I” in “Iseq” are never explained.

⁶The tutorial says it was written in Miranda, but I am not sure if this is correct. All of the code samples run successfully in Haskell, and I believe that Miranda’s syntax is somewhat different.

2 Background

This section provides brief background on programming language theory from a functional perspective.

2.1 Definition of a programming language

Interfaces are necessary for efficient and effective communication. A *programming language* serves as an interface between humans and computers. To rigorously work with a programming language, we define its *syntax* and *semantics*.

The syntax of a programming language describes the way valid expressions and programs are formed. It is specified using a *grammar*. For example, the grammar for the untyped λ -calculus Λ is shown in Figure 1.

The *semantics* of a programming language describes its behavior. The *static semantics* denotes the behavior of processes that happen prior to evaluation, such as type checking. The *dynamic semantics* describes the behavior of *evaluation*. Evaluation is the process of reducing an expression down to a *value*, or irreducible expression.

For this project, we define two languages: the Core language, and the abstract stack-based language of the G-machine. The semantics of the Core language is, like many functional languages, highly based on the λ -calculus. The semantics of the stack-based G-machine language is similar to other stack languages such as Forth.

2.2 Implementation(s) of a programming language

A programming language is a specification. There may be multiple *implementations* of a language. The specification and implementations are orthogonal. A single language may have multiple implementations with different performance characteristics and language support. For example, the Core language has four implementations proposed in the tutorial (TI, GM, TAM, PGM); other major languages such as the JVM, Python, or Haskell also have multiple implementations.

We typically classify implementation into one of two broad classes: *interpreters* (a.k.a. *evaluators*) or *compilers*. The difference is that interpreters take a program's source code and runs it "directly." On the other hand, a compiler performs a two-stage process: compiling to a low-level representation, to be evaluated at some later time. Compiled representations tend to perform better and require simpler *runtimes*. The distinction between interpreters and compilers may not always be clear; in some cases, they form a spectrum delineated by the complexity of the runtime necessary for program evaluation.

In this project, we will examine two separate implementations of the Core language. The first implementation directly encodes the idea of graph template instantiation and reduction. Since all of the graph operations occur within Haskell code (a Haskell runtime), this is an interpreter. We improve this with the G-machine by compiling graph construction to a series of opcodes that may

$$e ::= \lambda x.e \mid x \mid e\ e$$
Figure 1: Grammar of Λ

run on a simpler stack-based machine, which may be implemented on many computer architectures with a simpler runtime⁷.

2.3 The untyped λ -calculus

The untyped λ -calculus Λ is a very basic model of universal computation⁸ proposed by Alonzo Church in the 1930's. The grammar of Λ is incredibly simple: it is given by Figure 1. We only have three expression forms in Λ : variables (bound by functions), λ -abstractions (functions of one variable x and return the expression e); and function application $e_1\ e_2$, in which e_1 is in function position and e_2 is in argument position. Note that function application is represented using spaces, and parentheses are only necessary to specify order of operations.

The dynamic semantics are also very simple, illustrated by the rules shown in Figure 2 using a big-step operational semantics. Without diving deep into the syntax, this means that λ -abstractions evaluate to themselves, and the application of e_1 to e_2 involves (recursively) evaluating e_1 to a λ -abstraction $\lambda x.e'_1$; substituting e_2 for every instance of x in e'_1 ; and then recursively evaluating that result. This is a very simple way to think about function application, and it can be shown that any data flow computation can be abstractly represented using this minimalist representation⁹.

The λ -calculus serves as the logical and notational basis for much of functional programming, including Haskell and Core. For example, using space as the function application operator originates from the λ -calculus. Also, lazy evaluation and pure functions mesh very nicely with the more mathematical framework that the λ -calculus provides, as opposed to the stateful model of imperative programming languages.

However, since Λ is so minimalist, it is not practical nor efficient. Usually we extend Λ with static typing, resulting in the simply-typed λ -calculus (STLC). In our case, we add a base type (integers) and structured data, and extend the grammar with `let` and `case` expressions, but do not implement static typing (which runs us the possibility of run-time type errors).

⁷Chapter 4 of the tutorial, which is not implemented for this project, uses a three-address machine, which may require an even simpler runtime on a three-address architecture such as x86-64.

⁸I.e., Λ can simulate a Turing machine.

⁹See Church notation.

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e'_1 \quad [e_2/x]e'_1 \Downarrow e}{e_1 e_2 \Downarrow e}$$

Figure 2: Dynamic semantics of Λ

2.4 Functional programming

Functional programming is a programming paradigm highly involved with function application, function composition, and first-class functions. It is a subclass of the declarative programming paradigm, which is concerned with pure expression-level computation. Imperative programming is often considered the complement of imperative programming, which may be characterized as programming with mutable state, side effects, or statements. Purely functional programming is a subset of functional programming that deals solely with mathematically-pure functions; non-pure languages may allow varying degrees of mutable state but typically encourage the use of pure functions. Miranda, Haskell, and Core are examples of non-strict (lazy) pure functional languages.

There are a number of memes¹⁰ among functional languages that may not be familiar to the imperative programmer. Some of these memes include (but are not limited to):

Spaces for function application This is a notational meme from Λ .

Immutability (i.e., lack of state) is an integral part of functional programming that allows for many optimizations such as structural sharing¹¹ or fast structural equality checking.

No side-effects Pure functional languages additionally disallow side-effects (stateful actions) such as printing or array-based operations, unless specifically encapsulated. This may make debugging via printing difficult, so we need a special `Print` opcode in Core.

Curry-by-default functions and partial application In some functional languages, functions of multiple variables may be *partially applied*. For example, consider the following Haskell program.

```
sum x y = x + y
add1 x = sum 1
main = add1 3
```

This may seem to be an error, since `sum` is only applied to one argument in the definition of `add1`. However, this is allowable because `sum` will not be invoked until it is applied to the right number of arguments, which will happen after applying `add1` to 3. If we treat function application

¹⁰In the original sense of the word, not in the Internet sense of the word.

¹¹We implement structural sharing for integer nodes.

as substitution, this may be more clear: `add1 3` may be substituted to `(sum 1) 3 ≡ sum 1 312`. Alternatively, we may think of any multi-arity function as being a series of nested λ -abstractions of one argument. The following is an equivalent definition of `sum`.

```
sum = \x . \y . x + y
```

Algebraic datatypes (ADTs) In Haskell, you can create type definitions such as the following:

```
data ListNode a = Nil | Cons a ListNode
data TreeNode = Leaf | Node Int TreeNode TreeNode
```

In this example, we see that we can very naturally describe linked lists using a recursive data structure with one of two forms: the base case `Nil`, or a `Cons` cell comprising a value and another (recursive) `ListNode`. This `ListNode` datatype is also parametrically polymorphic with type parameter `a`. We see a similar case for the (non-parametric, recursive) definition of `TreeNode`. ADTs allow us to naturally and safely express composite data and variant types, and language constructs such as pattern-matching and `case` expressions allow for safe deconstruction of structured data. In Core, we will implement a simpler-to-implement system of data constructors than user-defined data constructors as shown in the example above, but it is equally as powerful.

2.5 Terminology

The tutorial assumes a background in (lazy) functional programming. Instead, I assume a background in imperative programming and a basic background in programming language compilers. Below are some terms that may be useful for understanding Haskell, Core, and this project in general. These terms are fairly standard and can be easily looked up on Wikipedia or Stack Overflow, but the following are my own definitions. I thought I'd include them here to avoid confusion, since some of them have somewhat unassuming names.

Lazy evaluation A general term that describes evaluation that only occurs when the expression is used. For the purposes of this paper, it refers to *normal-order* evaluation, and more particularly *call-by-need* evaluation. Note that laziness (and eagerness) can refer to

Eager evaluation The opposite of lazy evaluation: evaluation that occurs when an expression is encountered. The eager analogue to normal-order evaluation is *applicative-order* evaluation.

Normal-order evaluation Lazy function application. In other words, the arguments to a function are not evaluated before the function body is

¹²These are equivalent due to the operator precedence and associativity of the infix function application operator (space).

evaluated. This is usually a feature only of pure functional languages such as Haskell or Core.

Applicative-order evaluation Eager function application. In other words, the arguments to a function are evaluated before the function body is evaluated. This is the way that most languages are implemented, such as C or OCaml.

Strict/non-strict More or less synonyms for eager/lazy. These terms are usually used when referring to Haskell's normal-order evaluation. Haskell allows you to control strictness using special operators¹³. We explore controlling strictness with strict/non-strict compilation contexts in Section 5.4.1.

Call-by-value More or less synonymous with applicative-order evaluation.

Call-by-name A naive form of normal-order evaluation. The unevaluated expression assigned to the function argument is lazily substituted into each instance of the argument, and evaluated when encountered.

Call-by-need A better form of normal-order evaluation, in which evaluation of arguments are memoized. Haskell and Core are call-by-need. This requires node updates and indirections as described in Section 4.2.5.

Supercombinator The top-level expression construct in Haskell and Core. Roughly equivalent to a function definition. All supercombinators are always in scope, including from the body of the supercombinator or before it is declared, allowing for easy simple and mutual recursion.

λ -abstraction Also known as λ -function, anonymous function, arrow function, function object, or simply function, in a language with first-class functions. In the λ -calculus, these typically take exactly one argument; however, in the case of Core or Haskell supercombinators or anonymous functions, these may take multiple arguments.

Function application Another name for function invocation, usually used in the context of functional programming. Note that functions may be *partially applied* in languages that support *currying-by-default*.

Currying A notational convenience in which a function of n arguments may be *partially-applied* by only calling it with $m < n$ arguments. This partially-applied expression itself is a (curried) function of $n - m$ arguments. When the function is applied to all n arguments, then the function is applied normally. This notation fits nicely into the interpretation of nested one-argument λ -abstractions, and is nicely handled by the TI and GM machines.

¹³https://wiki.haskell.org/Performance/Strictness#Explicit_strictness

Currying-by-default A property of a programming language in which functions of multiple arguments may be curried (partially applied). Haskell, OCaml, and Core are examples of such languages. In languages without currying-by-default, currying may be achieved using nested one-argument λ -abstractions.

Product types Composite datatypes, e.g., structs (a.k.a., objects, `records`, or named `tuples`) or tuples.

Sum types Disjunctive datatypes, notably *tagged unions*.

Algebraic datatypes (ADTs) Composite datatypes allow for arbitrary compositions of sum (tagged union) and product (tuple) types. These are a common construct available in many strongly-typed functional languages, such as Haskell, OCaml, and Rust. These are safely deconstructed using `case` expressions and pattern-matching. Data that is formed from ADTs are known as *structured data*.

Weak head normal form (WHNF) An expression is in WHNF if the outermost expression is a data constructor or a λ -abstraction. In other words, it defines what a fully-evaluated expression is in Core. Alternatively, we can think of it as the canonical representation of a datum in Core. (There are also normal and head normal forms, which are not relevant to Core.)

Scrutinee The expression to be matched in a `case` expression. In the case of the Core language, this must be structured data, but it usually is any pattern to match against.

Definiens The expression that a variable is bound to.

Opcode A.k.a. instruction. Simple statement in an imperative assembly-like language.

Runtime For the purposes of this project, usually refers to the run-time evaluator for a programming language implementation. As opposed to “run-time,” which is an adjective meaning “occurring at evaluation time.”

Precedence	Associativity	Operator
6	left	application
5	right	*
5	none	/
4	right	+
4	none	-
3	none	=, ~, >, >=, <, <=
2	right	&
1	right	

Table 2: Core infix operator precedence and associativity

3 The Core language

3.1 Syntax

The grammar of Core is given in Figure 3. This is reproduced from Figure 1.1 in the tutorial. λ -abstractions are omitted from the syntax, since they are only introduced in Chapter 6 with the introduction of the λ -lifting program transformation.

The precedence of infix operators is provided in Table 2, reproduced from Table 2 in the tutorial. Note that unary negation is not provided as a primitive operator instead, the primitive unary function `negate` is provided, and it behaves like a normal function.

Core may be thought of as Λ extended with number types, arithmetic and boolean operators, structured data, `let(rec)` expressions, and `case` expressions. The top-level of a program is simply a sequence of semicolon-delimited supercombinator definitions.

3.2 Dynamic semantics

The dynamic semantics of basic function application in Core is the same as Λ , taking into account currying. The rest of the expression forms should be fairly clear, and are familiar for those acquainted with Haskell or other functional languages with local definitions, algebraic datatypes, and pattern matching.

The expression forms to note are `let(rec)` expressions, `case` expressions, and data constructors using the `Pack(t,a)` expression form. `let` expressions allow for a series of local variable bindings. `letrec` expressions allow for a series of local variable bindings, but all of the variables being defined are in scope of the other variables being defined (e.g., allowing for mutually-recursive definitions)¹⁴. `Pack` and `case` are used to construct and destruct structured data, respectively, in a simple form without introducing user-defined data con-

¹⁴Neither `let` nor `letrec` expressions are too important for a language like Core. The same functionality may be implemented with supercombinator definitions (including mutual recursion), but local definitions afford greater convenience.

```

program ::= sc1 ; ⋯ ; scn                                (top-level program)
sc ::= fn var1 ⋯ varn = expr                         (supercombinator)
expr ::= expr aexpr                                         (application)
       | expr1 binop expr2                               (infix binary application)
       | let defns in expr                                 (local definition(s))
       | letrec defns in expr                            (local recursive definition(s))
       | case expr of alts                             (case expression)
       | aexpr                                            (atomic expression)
aexpr ::= var                                              (variable)
        | num                                             (number (integer))
        | Pack{num, num}                                  (data constructor)
        | ( expr )                                       (parenthesized expression)
defns ::= defn1 ; ⋯ ; defnn                                (definitions)
defn ::= var = expr                                       (definition)
alts ::= alt1 ; ⋯ ; altn                                (case alternatives (rules))
alt ::= <num> var1 ⋯ varn -> expr                (case alternative)
binop ::= arithop | relop | boolop                      (binary operators)
arithop ::= + | - | * | /                               (arithmetic ops)
relop ::= < | ≤ | == | ~= | ≥ | >                   (comparison ops)
boolop ::= & | ||                                     (boolean ops)
var ::= alpha varch1 ⋯ varchn                      (variables)
alpha ::= an alphabetic character                     (identifiers 1)
varch ::= alpha | digit | -                          (identifiers 2)
num ::= digit1 ⋯ digitn                           (numbers)

```

Figure 3: BNF syntax for the Core language

```

data ListNode = Nil | Cons Int ListNode
length xs = case xs of
  Nil -> 0
  Cons _ xs' -> 1 + length xs'

main =
  let lst = Cons 2 (Cons 3 (Cons 4 Nil))
  in length lst -- returns "3"

-- Nil is represented with Pack{3, 0}
-- Cons(x, y) is represented with Pack{4, 2}
length xs = case xs of
  <3> -> 0 ;
  <4> x xs -> 1 + length xs

main =
  let lst = Pack{4,2} 2 (Pack{4,2} 3 (Pack{4,2} 4 Pack{3,0}))
  in length lst -- returns "3"

```

(a) Structured data in Haskell

(b) Structured data in Core

Figure 4: Structured data usage

structures such as in Haskell. Figure 4 demonstrates the use of both of these forms with an implementation of `length`, a function to compute the length of a linked-list ADT.

3.3 Base definitions

We adopt the conventions shown in Figure 5. These represent canonical/standard representations of booleans, lists, and the `if` expression using previously-defined expressions¹⁵. Note that different structured data do not necessarily need to have globally distinct tags; only different variants of the same ADT need be unique. For simplicity, we make all of the ADTs have unique tags, and reserve tags 1 through 4 for these¹⁶.

This forms part of our *prelude* (builtin definitions) for Core, and these definitions may be overridden at any time. Additionally, we provide a sample standard library of useful definitions, such as common combinators, arithmetic aggregation operators, list/stream operations, etc. in the file `examples/gprelude.core`.

¹⁵My conventions are slightly different from the textbook's ones, but this is only really a matter of style.

¹⁶This is a soft restriction.

```

False = Pack{1, 0} ;
True = Pack{2, 0} ;
Nil = Pack{3, 0} ;
Cons = Pack{4, 2} ;
if scrut t f =
  case scrut of
    <1> -> f ;
    <2> -> t

```

Figure 5: Standard representations of structured data and `if` expressions

We call this the standard library or extended prelude. These may be found in Appendix A.1.

3.4 Sample programs

Sample Core code may be found in Appendix A.

3.5 Lexer

Tokenization (lexing) of the Core language is fairly simple. There are almost no reserved keywords, except `Pack`, `let`, and `case`, and the number of expression types is small. Identifiers and numeric tokens are easy to identify using the grammar rules. Other symbols (except whitespace, which is discarded) are counted as one- or two-letter operators. Lines beginning with `--` are treated as comments and discarded¹⁷.

3.6 Parser

The parser described in the tutorial is a fairly simple LL(1) recursive-descent parser, built from scratch. Following the functional way¹⁸, this is built up from a set of primitive subparsers in a very composable way. Each (sub)parser is a function that takes a set of tokens as input, and returns a set of transformed matches and remaining tokens. We first define the primitive subparsers, which may recognize forms such as numeric literals, identifiers, or a specified string. Then, we develop two compositional parsers, `pAlt` and `pThen`. `pAlt` takes two subparsers and returns a parser that will match either subparser (and thus may return multiple matches). `pThen` takes two subparsers and matches only if the first subparser matches, and then the second subparser matches on the remaining tokens. By composing these generic subparsers, we build up a parser for Core. The top-level parser parses the whole program and returns the first successful parse. There may be multiple parses due to the *dangling-else problem*.

¹⁷This is the Haskell comment syntax.

The tutorial goes into significant detail about practical considerations when implementing a parser, such as *left recursion* (to prevent infinite recursion), implementing infix operator precedence and associativity (by splitting the *expr* production into separate productions for each precedence level), and the *dangling else problem* (which is a parsing ambiguity that arises in *case* expressions). For brevity, I omit a thorough discussion of these topics.

3.7 Pretty-print utility

The tutorial describes a pretty-print utility that is useful for performance and presentation purposes. This is not only useful for printing expressions, but also for any other debugging output, such as printing the program state.

A naive approach to printing information would be simply performing string concatenation. However, this is a $O(n)$ process on each string concatenation (and thus roughly $O(mn)$ with a large number of concatenations m), and not efficient for very large outputs. Additionally, we would like the ability to easily indent chunks of the formatted output. This is where the **ISeq** data structure comes in¹⁸. Stringbuilders objects (e.g., Java's `java.lang.StringBuilder`) are able to efficiently concatenate strings, but do not have the ability to provide the indentation layout that we desire.

ISeq works by storing the built string as a tree of appended strings. The string concatenation process then becomes the $O(n)$ process of creating an **Append seq1 seq2** node with the two subsequences. Additionally, indented blocks may be created using the **Indent seq** node, which takes a subsequence to indent at the current column number. As a result, building a formatted string is very efficient, since each of these operations is constant time.

The **ISeq** is then *rendered* to a string using the **IDisplay** method. This performs the single string concatenation. This method implements indentation by always keeping track of column count. Rendering the **ISeq** is a $O(n)$ operation as desired.

The **ISeq** data structure is very general and may be used for pretty-printing purposes outside of this project.

¹⁸I believe the “I” stands for “indentation,” but I do not know. I have asked on Stack Overflow but have not received a response.

4 Template instantiation (TI) evaluator

Template instantiation is only briefly reviewed in the tutorial. A more full description is given in Chapters 11 and 12 of “The implementation of functional programming languages” [6]. I will do my best to provide an intuitive high-level overview.

4.1 Intuitive walkthrough of a TI evaluation

The principal idea behind lazy evaluation is that an expression is only evaluated when necessary. We represent an expression using a graph, and evaluate the program by repeatedly reducing the graph until we are left with a data value (a final program state).

For now, let us consider two sample programs, shown below.

```
-- Program A
f x = x + 1
main = f 2

-- Program B
square x = x * x ;
main = square (square 3)
```

We may represent any supercombinator using a graph. For example, the graphs for `square` and `main` from Program B are shown below. The `o` symbol is used to represent function application. Due to automatic function currying, the evaluation of a multi-arity function application `x y z` actually is implemented as a series of multiple function applications of one variable, i.e., `((g x) y) z`. Infix binary operators appear as functions of two variables in the call graph. Also note that if the same variable is referenced multiple times in the graph, the nodes will point to the same instance – this is important for call-by-need evaluation.

```
-- square
    o
    / \
    o   \
    / \---x
    *
-- main
-- Program B state 0
    o
    / \
    /   o
    /   /
square  3
```

There are two types of reductions. We may either reduce a supercombinator (function) application, or a primitive operation. The reduction of a supercombinator means replacing the graph with the supercombinator and its arguments with the supercombinator's graph. The reduction of a primitive (e.g., arithmetic, relational, or boolean operators) means performing the primitive operation.

Consider Program A for an example of a supercombinator reduction. The graphs for `main` and `f` are shown below.

```
-- f
  @
  / \
  @   1
  / \
+   x
-- main
-- Program A state 0 (initial program state)
  @
  / \
f   2
```

The program begins with the graph of `main`. Clearly, we have to reduce `f` next. We do this by *instantiating* the graph of `f`, and substituting the arguments (in this case, $x \mapsto 2$) in the graph. Thus the next evaluation state would be:

```
-- Program A state 1
  @
  / \
  @   1
  / \
+   2
```

At this point, we see that the next reduction would be the application of `+` to arguments `2` and `1`, i.e., $(+ 2) 1$ or $2 + 1$. We use the underlying addition operator in Haskell, and replace the addition operation with the result. At this point, there are no more redexes in the program, so we are done and the program terminates.

```
-- Program A state 2 (final program state)
3
```

The TI always chooses the outermost reducible expression (redex) to reduce; this evaluation order is called *normal-order evaluation*. In other words, if we consider Program B again, we see that there are two redexes that we can reduce from the initial program state: the outer or the inner `square`. If we reduce the inner `square` first, then this is *applicative-order* (strict) evaluation. However,

we wish for *normal-order* (lazy) evaluation, so we reduce the outer expression first.

The way we find the outermost supercombinator or primitive is by walking the left-branch of application nodes; this list of nodes is called the *spine* of the graph. The spine will be stored on the stack; we start from the root, and push the application nodes of the spine onto the stack until we reach a supercombinator or primitive. The spine (and thus the stack) also contains the list of arguments passed to a supercombinator or primitive. With this in mind, the next program state would be:

```
-- Program B state 1
    0
    / \
    0   \
    / \---!0
*      / \
square   3
```

At this point, we would like to reduce the `*`, but there is a problem. The arguments have not been evaluated yet. Thus a primitive operator is only a redex if all of its arguments have been evaluated (to WHNF). This is not the case for this program: the root of our next redex is the node marked `!0`. Note that this isn't on the spine, so we create a new temporary graph with which to evaluate the arguments before returning to the evaluation of the `*` primitive. The node marked `!0` becomes the root of our new graph, and the old graph is saved.

```
-- Program B state 2
    0
    / \
square 3

-- Program B state 3
    0
    / \
    0   \
    / \---3
*
-- Program B state 4
9
```

The multiplication in state 3 is a redex since both arguments are fully evaluated to WHNF. Thus, this reduces down to a number. Now, we are not done, since we still have to finish the initial graph. Thus, we "pop" the old graph from state 1 off of the "stack of old graphs," and replace the unevaluated argument at the node marked `!0` with the evaluated result, 9.

```
-- Program B state 5
  0
 / \
  8   \
 / \---9
*
-- Program B state 6
81
```

At this point, there are no more “old graphs,” so evaluation terminates and we have the result.

This section is intended to give a high-level overview of the TI evaluator. The following sections will provide more clarity and be more precise.

4.2 The TI abstract machine

The TI machine is represented using an *abstract state machine*, and evaluation is described using *state transitions*. The state comprises a *stack*, a *heap*, and a *dump*.

The stack holds the spine of the (current) graph. We use this to find the next redex during the unwinding process, and to retrieve the list of arguments for a primitive or supercombinator. This will be discussed in Sections 4.2.1 and 4.2.2. The heap stores all of the (automatically-allocated) nodes used throughout. This will be discussed in Sections 4.2.4 and 4.2.5. The dump is a stack of stacks, and stores the “old graphs” or “old spines” when we need to strictly evaluate an argument, as shown in the Program B example in Section 4.1.

The TI evaluation process alternates between unwinding and instantiation. We will describe these in Sections 4.2.1 and 4.2.2.

4.2.1 Unwinding

Unwinding is the main evaluation semantics of the TI abstract machine. It dispenses an action based on the element on the top of the stack, and thus is easy to model using a state transition machine¹⁹.

It is the process of finding the next redex and choosing which action to perform. In the simplest form, it works by traversing the spine until we find a supercombinator. The state transition rules are very simple. If we encounter a number, the spine of the stack is empty, then evaluation is complete and the program terminates. If we encounter a function or application node, we push the application node onto the stack and continue unwinding the left node (continue unwinding the spine). Lastly, if we encounter a supercombinator node, then we instantiate its body using the arguments that are currently on the stack.

¹⁹We will see that this is not the case of instantiation, which puts it at odds with the state transition formalization. However, the G-machine will compile the instantiation step into a series of small, relatively-atomic opcodes.

This gets more complicated when primitive operators are introduced. If we encounter a primitive operator, we first check if the argument node(s) are evaluated to WHNF. If they are already evaluated, then the primitive operator is a redex and we perform the operation. If an argument is not a redex, then we push the current stack onto the dump, and set the argument as the singleton element of the new stack, and use this to evaluate the argument. Once the argument is fully-evaluated, then we pop the old stack from the dump, and resume evaluation of the primitive operator.

This is similar to you using a call stack. In fact, the dump is very similar to using a call stack, and this parallel will be accentuated when we encounter the `Eval` opcode in the G-machine, which performs the analogous operation. Moreover, the dump may be implemented like the call stack by pushing and popping *stack frames*; we do not need a separate stack data structure. The difference is that stack frames on the dump are only created when we need to strictly evaluate an argument, rather than every time a function is invoked as is the case for a call stack; this naturally allows for tail-call optimization (TCO)²⁰.

4.2.2 Instantiation

Instantiation is the process of creating a supercombinator graph when a supercombinator is applied. The process is fairly intuitive from a high-level perspective (e.g., it is visually clear from the examples in Section 4.1 how instantiation works). We may say that instantiation involves copying the graph structure of a supercombinator, and falling in any instances of its argument variables using the parameters used to invoke the supercombinator.

Instantiation is also fairly easy to implement in native Haskell code. It involves recursing through the Core language expression (represented as an AST), and building the corresponding call graph out of variable, numeric literal, and application nodes in the heap.

As mentioned in an earlier footnote, instantiation is at odds with the state machine representation because it is largely an atomic action and cannot be easily decomposed into simpler steps to run on a simpler runtime. The main difference between the TI and the G-machine implementations is that the instantiation step (the graph creation) is compiled to a series of opcodes in the G-Machine. The G-machine also allows us to discard the AST representation after compilation, since we don't need the instantiation function which operates on Core language expressions.

4.2.3 Advanced instantiations

So far, we have covered the instantiation of the core forms from A: functions, variables, function application, and integers. These are covered in Mark 1 of

²⁰TCO allows for arbitrary recursion depth if a function is tail-recursive. This is not true of regular call-stack-based languages, since each recursive call will generate a new stack frame and quickly exhaust the stack. However, it can be implemented on a call stack using techniques such as trampolines, but this is much messier and less elegant than our natural TI interpretation of graph reduction.

the TI evaluator. There are additional expression forms dealt with in Marks 2 and 5; these are described below in brief.

Mark 2 introduces the instantiation of `let(rec)` expressions. This involves instantiating each of the definitions and augmenting the environment. The recursive form involves evaluating the definitions in an environment that includes the `letrec` bindings. This is a relatively straightforward task.

Mark 5 introduces the instantiation of structured data. However, there is difficulty instantiating `case` expressions²¹, so a limited form of destructuring using builtin functions `caseList`, `casePair`, and `if` are used to provide conditional statements.

4.2.4 Memory model

The memory model of the TI is very simple. The heap is simply an abstract data structure mapping addresses to nodes. The interface for this data structure has methods to allocate, lookup, and free nodes. Nodes are automatically allocated on demand. Nodes are never freed in my implementation, since I did not implement garbage collection; however, the `mark-scan` and `two-space` garbage collection methods are developed at the end of the chapter. This heap data structure will remain the same for the G-machine implementation.

4.2.5 Updates and indirections for call-by-need evaluation

Updates and indirections are two topics that are necessary to implement efficient call-by-need normal-order evaluation. Without updating nodes after they have been evaluated, repeated usages of a node will result in repeated evaluations, resulting in an inefficient call-by-name normal-order evaluation. When a node is updated, it is replaced with an *indirection node* (essentially, a pointer or reference) to the evaluation result. The indirection node is a necessary part of the updating to prevent nodes from being copied (and thus inefficiently evaluating multiple times). This is implemented in Mark 3 of the TI evaluator. The motivation for updating and indirection nodes is provided in section 2.1.5 of the tutorial, and we do not reproduce it here since it is fairly intuitive.

²¹This is exercise 2.18 in the tutorial. I am actually not too sure what the answer should be, and I have posted my best guess as a Stack Overflow question and answer. My guess is that the code is similar to deconstructing structured data at instantiation-time (which is separate from evaluation-time), and that each of the specialized deconstruction methods have "workarounds" for this problem.

5 G-Machine (GM) implementation

The G-Machine is an abstract machine with a very similar idea to the TI machine with graph reduction and unwinding. The difference is that the graph creation which is a complicated process performed by the instantiation step of the TI machine is converted into a series of opcodes on a stack-based machine. The unwinding step remains largely the same. The state of the machine is largely the same, with a stack, dump, and heap. Notably, this also includes an instruction queue (which replaces instantiation), and supercombinators are compiled to a sequence of instructions.

This stack-based machine is desirable for several efficiency reasons: it requires a simpler (smaller) runtime; it does not require the re-traversal of a supercombinator's body expression on each expression; and the runtime does not require an expression-level representation of the language.

5.1 Sample compilation

We first try to develop an intuition of the compilation process. The example below is given in section 3.1.1 of the tutorial.

```
f g x = K (g x)
This compiles down to the following opcode sequence.
```

```
Push 1
Push 1
Mkap
Pushglobal K
Mkap
Update 3
Pop 3
Unwind
```

These instructions should be indicative of a stack-based machine. We can envision the state of the program after each instruction. Assume that the supercombinator was invoked like so: `f I 2`. Thus we expected `I` and `2` to be already pushed onto the stack.

Note that this uses the updated addressing mode described in Section 5.3. Thus the stack does not initially contain the application spine, but rather only the root of the supercombinator application and the arguments, pushed in reverse order.

We omit the dump from this evaluation trace; it is not relevant here, since there are no primitive operators.

```
-- State 0: initial program state (upon applying f I 2)
Instruction queue: Push 1; Push 1; Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind
```

```

Stack:
0. #0
1. #2
2. #4

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main

Node 0 represents the root of the supercombinator application, which is
represented by the following graph:

Graph(s):
    #0(0)
    /   \
    #3(0)   #2(2)
    /   \
    #3(f)   #4(I)

We will now walk through the instruction sequence. The effect of each
instruction should be relatively straightforward, since they are relatively simple
stack instructions.

-- State 1: after (first) Push 1
Instruction queue: Push 1; Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #2

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main

```

```

Graph(s):
    #0(0)
    /   \
#3(0)   #2(2)
    /   \
#3(f)  #4(I)

-- State 2: after (second) Push 1
Instruction queue: Mkap; Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #2
4. #4

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main

Graph(s):
    #0(0)
    /   \
#3(0)   #2(2)
    /   \
#3(f)  #4(I)

-- State 3: after (first) Mkap
Instruction queue: Pushglobal K; Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #7

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2

```

```

#3 NSupercomb f
#4 NSupercomb i
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2

Graph(s):
    #0(0)
    /   \
#3(0)   #2(2)
    /   \
#3(f)   #4(i)

#7(0)
    /   \
#4(i)   #2(2)

-- State 4: after Pushglobal K
Instruction queue: Mkap; Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #7
4. #5

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2

Graph(s):
    #0(0)
    /   \
#3(0)   #2(2)
    /   \
#3(f)   #4(i)

#7(0)
    /   \

```

```

#4(I) #2(2)
-- State 5: after (second) Mkap
Instruction queue: Update 3; Pop 3; Unwind

Stack:
0. #0
1. #2
2. #4
3. #8

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
    #0(8)
    /   \
    #3(8)   #2(2)
    /   \
    #3(f)   #4(I)

    #8(8)
    /   \
    #5(K)   #7(8)
    /   \
    #4(I)   #2(2)

-- State 6: after Update 3
Instruction queue: Pop 3; Unwind

Stack:
0. #8
1. #2
2. #4
3. #8

```

```

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
    #0(0)
    /   \
#3(0)   #2(2)
/   \
#3(f)  #4(I)

#8(0)
/   \
#5(K)  #7(0)
/   \
#4(I)  #2(2)

-- State 7: after Pop 3
Instruction queue: Unwind

Stack:
0, #8

Heap:
#0 NAp #1 #2
#1 NAp #3 #4
#2 NNum 2
#3 NSupercomb f
#4 NSupercomb I
#5 NSupercomb K
#6 NSupercomb main
#7 NAp #4 #2
#8 NAp #5 #7

Graph(s):
#8(0)
/   \
#5(K)  #7(0)
/   \

```

#4(I) #2(2)

As we can see, the graph constructed by the compiled opcodes produces the result as instantiation.

The last instruction is the most complicated. This is the `Unwind` opcode, which corresponds to the unwinding action discussed in the TI evaluator (setting up the instructions for the next instantiation). Up until now, all of the instructions have been used to create the graph that would have been created through instantiation. After the `Unwind` opcode, we expect to have a setup similar to state 0 for evaluation to continue again.

5.2 List of opcodes

The list of opcodes of the abstract stack machine that the G-Machine compiles to is given below. For the sake of this document, the dynamic semantics of each opcode is only provided informally²². A precise definition of most of the opcodes are given in the tutorial²³.

Pushglobal *f* Push the address of the global (supercombinator) *f* onto the stack²⁴.

Pushint *n* Push the address of an integer node representing the integer *n* onto the stack. If such an integer node representing the integer *n* already exists, this may use that existing address; otherwise, this will allocate a new integer node *n*.

Push *n* Push the *n*-th address of the stack onto the stack.

Update *n* Set the node pointed to by the *n*-th address of the stack to an indirection node pointing to the top element of the stack, then pop one element from the stack.

Pop *n* Pop *n* elements from the stack.

Alloc *n* Push *n* invalid addresses onto the stack, or decrease the stack pointer by *n*²⁵.

Slide *n* “Slide” the top element of the stack up *n* slots. In other words, the (*n*+1)-th element of the stack will be replaced with the top of the stack, and then *n* elements will be popped.

²²The syntax of this assembly-like language is trivial and need not be stated.

²³Most of the time, the state transition(s) of an opcode are provided, and the implementation of the opcode in Haskell is left as an exercise.

²⁴This is described in Mark 6 for a technicality regarding unsaturated data constructors.

This is Exercise 3.38.

²⁵This is used for `letrec` expressions. These addresses initially point to an invalid node, and will be updated by the evaluation of the definitions.

Unwind Similar in function to unwinding in the TI implementation Section 4.2.1. Finds the next redex along the spine. Also performs the stack restructuring described in Section 5.3. When the supercombinator or primitive is found, adds the compiled opcodes to the instruction queue.

Mkap Pop two elements from the stack, allocate a new application node (with the first popped element in function position, and the second popped element in argument position) and push the application node onto the stack.

Eval Strictly evaluate the element on the top of the stack to WHNF on a new stack (frame)²⁶.

Add, Sub, M1, Div, Neg Arithmetic binary and unary operators. Assumes the two elements on the top of the stack are evaluated to integers (otherwise will crash the program). Pops the top two elements and leaves a number node on the top of the stack representing the arithmetic result.

Eq, Ne, Lt, Le, Gt, Ge Relational binary operators. Assumes the two elements on the top of the stack are evaluated to booleans in the standard representation (otherwise will crash the program). Pops the top two elements and leaves a node on the top of the stack representing the boolean result in the standard representation.

Pack $t\ n$ Pops n addresses from the stack. Allocate and push a structured data node representing structured data with tag t and the n popped arguments.

Casejump $rules$ The set $rules$ is of the form $\{t_i \rightarrow [i_1, i_2, \dots]\}$: a mapping of tags t_i to code sequences $[i_1, i_2, \dots]$. Assumes the top of the stack is the scrutinee of a **case** expression, and thus a data node with tag t . Appends the instruction list of the appropriate rule onto the current instruction queue. Throws an error if the scrutinee is not a structured data node or if there is no match.

Split n Assumes the top of the stack is a structured data node. Pops it from the stack, and then pushes its n arguments onto the stack.

Print Output the top element on the stack. Assumes that it is a data node (integer or structured data node). Recursively generates additional **Print** and **Eval** opcodes for structured data.

5.3 A simpler stack addressing mode

In the template instantiation, the spine was pushed onto the stack. This means that for a supercombinator with n arguments, the spine has n **Map** nodes, followed by the supercombinator node at the top of the stack. In order to get the m -th argument, we traverse up $(m+1)$ nodes, and have to perform an extra lookup on the **Map** to get the address of the argument.

²⁶This pushes a new stack on the dump and evaluates the node on the new stack. This is more or less equivalent to a function call in the conventional procedural ABIs.

Mark 3 of the G-machine proposes a simpler stack addressing scheme. To prevent extra indirections, we restructure the stack during an `Unwind` instruction so that only the arguments for a supercombinator are left on the stack, rather than the `Mmap` nodes. The exception is that the root of the supercombinator application redex is left on the stack, so that it may be updated with the evaluation result of the redex. This method extends naturally to `let(rec)` expressions.

This aligns with the x86 ABI, which involves pushing arguments onto the stack in reverse order. Of course, there is no notion of application nodes in a low-level language. The only difference is the root of the redex remaining under the arguments on the stack.

5.4 Compilation schemes

Compilation is divided into several **compilation schemes**. A high-level overview of the compilation schemes is shown below²⁷. A list of rules for these compilation schemes may be found in [3].

- SC* compilation scheme** Compile a supercombinator. Wrapper around \mathcal{R}
- R* compilation scheme** Strictly compile a supercombinator body using the \mathcal{E} compilation scheme, and then unwind the stack.
- C* compilation scheme** Compile an expression in a lazy compilation context.
- E* compilation scheme** Compile an expression in a strict compilation context.
- D* compilation scheme** Compile a `case` expression.
- A* compilation scheme** Compile an alternative in a `case` expression.

For sake of brevity, I will not describe all of the compilation schemes in detail. Instead, I will only discuss the compilation schemes for the Mark 1 G-machine, which should give a good idea of how the compilation schemes work. These are shown in Figure 6, which reproduces Figure 3.3 from the tutorial.

The rules for \mathcal{SC} and \mathcal{R} are simple; these are drivers for code generation for top-level supercombinators. The \mathcal{C} compilation scheme is the most interesting. We see how functions, variables, and integers are compiled. ρ is an environment mapping variable names to offsets on the stack²⁸. Global variables and numbers that appear in the expression are simply pushed onto the stack using the `Pushglobal` and `Pushint` opcodes. Local variables are looked up in the environment, and a `Push` opcode is generated. Lastly, function application is generated postfix-style. This compiles the argument, then the function. After these two are compiled, a `Mmap` instruction is generated to create the application

²⁷N.B. I have no idea why these evaluation schemes are named the way they are. There is no indication in the textbook, or anywhere online, as far as I can tell.

²⁸This is similar to `rbp`-relative addressing for local variables.

```

 $SC[f \ x_1 \ \dots \ x_n = e] = \mathcal{R}[e] [x_1 \mapsto 0, \dots, x_n \mapsto n-1] n$ 
 $\mathcal{R}[e] \rho d = \mathcal{C}[e] \rho \text{++ } [\text{Slide } d+1, \text{Unwind}]$ 
 $\mathcal{C}[f] \rho d = [\text{Pushglobal } f]$ 
 $\mathcal{C}[x] \rho = [\text{Push } (px)]$ 
 $\mathcal{C}[i] \rho = [\text{Pushint } i]$ 
 $\mathcal{C}[e_0 \ e_1] \rho = \mathcal{C}[e_1] \rho \text{++ } \mathcal{C}[e_0] \rho^{+1} \text{++ } [\text{Mkap}]$ 

```

Figure 6: G-machine Mark 1 compilation schemes

node from the top two elements on the stack. This covers the basic operations for the λ -calculus logic.

A full description of the full compilation schemes for the Mark 6 G-machine would be too long to include here. Mark 3 implements `let(rec)` expressions, which are straightforward but tedious. Mark 4 implements arithmetic (which introduces the `Eval` opcode to strictly evaluate an element on the top of the stack by using the dump). Mark 5 implements the strict \mathcal{E} compilation scheme described in Section 5.4.1. Mark 6 implements structured data, which is fairly complicated and has some nuances regarding strict compilation and unsaturated constructors that require program transformations or other tricks²⁹.

5.4.1 Non-strict vs. strict compilation context

Mark 4 of the G-machine introduces primitives into the language, and is the first iteration of the G-machine in which strictness becomes relevant. The ordinary compilation of an primitive operation such as addition produces the following opcode sequence.

```

Push 1
Eval
Push 1
Eval
Add
Update 2
Pop 2
Unwind

```

However, this seems like an awful lot of work to perform a simple addition operation. The book motivates this with the simple program:

```

main = 3+4*5

```

²⁹See section 3.8.7 or exercise 3.38 in the tutorial.

Ordinarily, this program will compile to the following program. In addition to this opcode sequence, each of the function applications will spawn the eight opcodes shown above.

```
Pushint 5
Pushint 4
Pushglobal "*"
Mkap
Mkap
Pushint 3
Pushglobal "+"
Mkap
Mkap
Eval
```

However, we can compile this to the following set of opcodes by inspection, which is much shorter, and doesn't include any function applications or `Eval` opcodes:

```
Pushint 5
Pushint 4
Mul
Pushint 3
Add
```

It should not be surprising that strict code may be more efficient than non-strict code. This gives rise to the \mathcal{E} evaluation scheme described in Section 5.4. However, we cannot always generate strict code, or else defeat the purpose of a lazy language. In particular, we cannot generate strict code for function arguments or definitions in a `let(rec)` expression, since these may never be evaluated and evaluating them strictly would make evaluation not normal-order.

The intuition behind the \mathcal{E} compilation scheme is that any expression that lies directly in a supercombinator body (i.e., not in a `let(rec)` definition or in a function argument) must be evaluated when the supercombinator is evoked. We compiled such expressions in the **strict evaluation context \mathcal{E}** . Other expressions are compiled in the **lazy evaluation context C** . Additionally, we do not need `Eval` opcodes when in the strict compilation context, since we can recursively deduce that all subexpressions are strictly evaluated.

Strict and lazy evaluation is purely for improvements in performance, and do not (should not) change the behavior of the program. The current mechanism to determine when a strict evaluation context is acceptable is very simple; the tutorial cites other research on the material such as [1]. Haskell also allows users to explicitly specify strict evaluation of an expression.

5.5 Evaluator

The evaluator implements the abstract stack machine. Each of the opcodes described in Section 5.2 is implemented as a separate function in the G-Machines's

Evaluator module.

5.5.1 Transpilation to x86-64 assembly code

To illustrate the simplicity of the G-Machine's runtime, the stack machine of the Mark 1 of the G-Machine was implemented in NASM x86-64 assembly code. Each stack machine opcode was implemented using an assembly macro. Super-combinator node definitions are also implemented with a macro. This can be found on GitHub at [jlam5555/fc-transpiler](https://github.com/jlam5555/fc-transpiler).

I did not have time to implement later Marks of the G-Machine in assembly code, and I believe that it is a reasonable project for future work. However, the increased complexity of later Marks will likely be conveniently handled by small helper functions written in C.

6 Future work

6.1 Garbage collection

For a functional language, garbage collection in the runtime is very important, since nodes are automatically allocated without control of the user. Garbage collection is covered in the last part of the TI implementation after Mark 6. It was not implemented for sake of time.

6.2 Three-address machine and parallel G-machine

These are two additional implementations of the compiler that are covered in future chapters of the tutorial. Both are very intriguing: the three-address machine is closer to the architecture of many modern CPU architectures, and inherently parallelized reduction is also an interesting idea.

6.3 Particulars of the STG and Haskell's implementation

It will be interesting to study the intricacies of Haskell's implementation. Haskell, despite being a high-level, lazy functional language, is known to be very fast to the point of bafflement³⁰, with optimized GHC-compiled programs usually only 2 to 5 times slower than gcc-compiled programs³¹.

We've seen that the graph-reduction mechanism in the TI machine is relatively intuitive, but there is significant complexity in the construction (instantiation) of graphs. The G-machine achieves some efficiency by avoiding the construction of a graph to code. It will be interesting to see additional optimizations that allow GHC to achieve its famed performance. While this tutorial is meant to be an introductory tutorial on such machines, it should be formative to read documents specifically about the design of Haskell, such as [4].

6.4 Compilation to native binaries

We achieved compilation of the Mark 1 G-machine to native binaries using assembly macros and the NASM assembler, as described in Section 5.5.1. We leave for future work the same work for the Mark 6 G-machine, which is significantly more complicated.

However, it may be more fruitful to attempt this after reading the chapter on the three-address machine, since this chapter should describe the translation from the stack-based machine to a three-address machine. Transpiling the three-address machine to x86-64 opcodes should also be a simpler process.

³⁰<https://stackoverflow.com/q/35027952>

³¹<https://benchmarksgame-test.pages.debian.net/benchmarksgame/fastest/ghc-gcc.html>

7 Conclusions

This tutorial teaches how to efficiently evaluate a pure lazy functional language. First, an intuitive method involving lazy graph reduction called the Template Instantiation (TI) evaluator is introduced. The runtime of the TI comprises alternating graph creation (instantiation) and reduction (unwinding) processes. The efficiency of the runtime is improved by compiling the graph creation process into a series of opcodes in an stack machine, called the G-machine.

For this project, I was able to complete both base implementations and achieve the desired results. There is still much of the tutorial that has not been implemented, such as garbage collection, the λ -lifting program transformation, and other implementations such as the three-address machine and the parallel G-machine.

8 References

- [1] Geoffrey Burn. *Lazy functional languages: abstract interpretation and compilation*. MIT Press, 1991.
- [2] Paul Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 12–1.
- [3] Thomas Johnson. "Efficient compilation of lazy evaluation". In: *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. 1984, pp. 58–69.
- [4] Simon L Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of functional programming* 2.2 (1992), pp. 127–202.
- [5] Simon L Peyton Jones and David R Lester. "Implementing functional languages: a tutorial". In: *Department of Computer Science, University of Glasgow* (2000).
- [6] Simon L Peyton Jones. *The implementation of functional programming languages*. Prentice-Hall, Inc., 1987.

A Code samples

A.1 Core standard library

These are a set of standard definitions that I found useful for exercises. They are not part of the “prelude,” i.e., the list of standard definitions builtin to the language as described in Section 3.3, so these definitions need to be loaded from a file.

```
-- Logical operators
and x y = if x y False ;
or x y = if x True y ;
xor x y = if x (not y) y ;
not x = if x False True ;

-- Useful simple combinators
id x = x ;
double x = x + x ;

-- Useful simple arithmetic operators and predicates
inc x = x + 1 ;
zero x = x == 0 ;
even x = x == ((x / 2) * 2) ;
odd = compose not even ;
mod m n = m - (m / n) * n ;

add acc val = acc + val ;
mul acc val = acc * val ;

-- Sample math functions
fac n = if (zero n) 1 (n * fac (n - 1)) ;
fib n = if (n < 2) n (fib (n - 1) + fib (n - 2)) ;

-- List operations, LISP-style
car l = case l of
  <>x y -> x ;
cdr l = case l of
  <>x y -> y ;
length l = case l of
  <> -> 0 ;
  <>x y -> 1 + length y ;
nth n l = case l of
  <> -> 0 ;
  <>x xs -> if (zero n) x (nth (n - 1) xs) ;
caar = compose car car ;
cadr = compose car cdr ;
```

```

cdar = compose cdr car ;
cddr = compose cdr cdr ;
caaddr = compose car caadr ;
caaar = compose car caaar ;
cdaadr = compose cdr caddr ;
cdcaa = compose cdr caaar ;
cadddr = compose car cdddr ;
cadar = compose car cdar ;
cdddr = compose cdr cdddr ;
cddar = compose cdr cdar ;

-- Sequences/streams
numsFrom x = Cons x (numsFrom (x + 1)) ;
nats = numsFrom 0 ;
ones = Cons 1 ones ;

-- List operations
map f lst = case lst of
  <*> -> Nil ;
  <*> x xs -> Cons (f x) (map f xs) ;
filter p lst =
  case lst of
    <*> -> Nil ;
    <*> x xs ->
      let rest = filter p xs in
      if (p x) (Cons x rest) rest ;
foldl f acc lst = case lst of
  <*> -> acc ;
  <*> x xs -> foldl f (f acc x) xs ;
take n lst = if (n <= 0) Nil (take2 n lst) ;
take2 n lst = case lst of
  <*> -> Nil ;
  <*> x xs -> Cons x (take (n - 1) xs) ;
takeWhile p lst = case lst of
  <*> -> Nil ;
  <*> x xs -> if (p x) (Cons x (takeWhile p xs)) Nil ;
drop n lst = if (n <= 0) lst (drop2 n lst) ;
drop2 n lst = case lst of
  <*> -> Nil ;
  <*> x xs -> drop (n - 1) xs ;

-- Aggregate operations
sum = foldl add 0 ;
prod = foldl mul 1 ;

-- Useful function to generate a range

```

```
range a b = map (add a) (take ((b - a) + 1) nats)
```

A.2 Project Euler 1

See Project Euler 1. This requires the standard library from Appendix A.1.

```
N = 1000 ;
ltN x = x < N ;

sumMultsOfMUnderN m =
  let multsOfM = map (mul m) nats in
  let multsOfMUnderN = takeWhile ltN multsOfM in
    sum multsOfMUnderN ;

main =
  let f = sumMultsOfMUnderN
  in f 3 + f 5 - f 15
```

A.3 letrec demonstration

The following Core program demonstrates the use of `lstrec`. It also demonstrates a Church encoding of pairs.

```
pair x y f = f x y ;
fst p = p K ;
snd p = p K1 ;
f x y =
  letrec a = pair x b ;
    b = pair y a
  in fst (snd (snd (end a))) ;
main = f 5 6
```

A.4 Infinite streams

It is easy to generate infinite streams using generator functions. Due to the laziness of Core, these only compute as many elements as are forced.

Alternatively, as in Haskell, it is easy to introduce infinite streams using by implicitly “tying the knot,” due to the simplicity of (mutual) recursion.

```
-- Natural numbers using a recursive generator function
numsFrom x = Cons x (numsFrom (x + 1)) ;
nats = numsFrom 0 ;

-- Tying the knot
ones = Cons i ones ;
```

A.5 Twice twice twice

This is a program that has boggled my mind for some time. It illustrates partial application in Core, and has a slightly unusual time complexity. This was inspired by Exercise 2.13 from the tutorial.

```
twice f x = f (f x)
inc x = x + 1

p1 = 0           -- returns 0
p2 = inc 0       -- returns 1
p3 = twice inc 0 -- returns ?
p4 = twice twice inc 0 -- returns ?
p5 = twice twice twice inc 0 -- etc.

main = p5        -- try p1, p2, ...
```

ECE491: Advanced Compilers Topics

Independent Study Proposal

Prof. Sable
Jonathan Lam
2022/01/20

1 Overview

The goal of this course is to explore the student's interests in advanced compilers topics not covered in ECE466: Compilers. The topic will be designing a compiler for a functional programming language, following a tutorial (listed in the references) by the creator of Haskell. Functional languages have been well touted for their many benefits over imperative programming, such as their use in compositional programs (which is a tenet of good software design), their use of immutable programming (another tenet of good design, and commonly used in formal program analysis), advanced type systems (e.g., the Hindley-Milner type inference system), and the high degree of optimization despite their high level of abstraction from the underlying architecture.

The workload will be almost completely self-study, following the tutorial. The tutorial is organized into six chapters: chapter 1 involves the definition of the functional language and data structures; chapters 2-5 involve different compiler implementations; and chapter 6 implements a useful language extension. Some of the necessary code is provided in the tutorial, and some is left as an exercise. After each major implementation (more or less after each chapter) a demo and explanation will be provided to the advisor.

2 Workload and deliverables

There will be weekly readings. The capstone project will be four different implementations for a compiler for the hypothetical language "Core" following the course materials. These compilers will generate low-level representations of the Core language that can be executed on well-defined abstract machines.

3 Resources

- Implementing Functional Languages: a tutorial (by the creator of Haskell and a Turing Award winner) <https://www.microsoft.com/en-us/research/wp-content/uploads/1992/01/student.pdf>
- Write you a Haskell. <http://dev.stephendiehl.com/fun/>

4 Timeline

Each chapter is roughly 40-60 pages and is estimated to take 2-3 weeks for self-study. The timeline below is designed with this in mind. Completing each chapter is a major milestone, and a brief demo and verbal explanation will be given after completing each chapter.

Weeks 1-2 Chapter 1: Introduction to the Core language and project setup

Weeks 2-3 Chapter 2: Template instantiation

Weeks 4-6 Chapter 3: The G-Machine

Weeks 7-9 Chapter 4: TIM: the three instruction machine

Weeks 10-12 Chapter 5: A Parallel G Machine

Weeks 13-15 Chapter 6: Lambda lifting

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

**Practical performance enhancements to the evaluation
model of the Hazel programming environment**

by Jonathan Lam

Friday, April 29th, 2022 @ 11AM
In-person: NAB 502
Zoom Meeting ID: 832 1233 4294
Zoom Password: 976935

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Engineering

Professor Fred L. Fontaine, Advisor
Professor Robert Marano, Co-advisor

Performed in collaboration with the
Future of Programming Lab at the University of Michigan
directed by Professor Cyrus Omar

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

**Practical performance enhancements to the evaluation
model of the Hazel programming environment**

by Jonathan Lam
Spring 2022

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Engineering

Professor Fred L. Fontaine, Advisor
Professor Robert Marano, Co-advisor

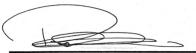
Performed in collaboration with the
Future of Programming Lab at the University of Michigan
directed by Professor Cyrus Omar

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.


5.11.2022
Barry L. Shoop, Ph.D., P.E. Date

 May 11, 2022
Fred L. Fontaine, Ph.D. Date

 2022-05-11
Robert F. Marano, M.Eng. Date

ACKNOWLEDGMENTS

I thank my advisor Professor Fred Fontaine for the ample feedback, even when submitting everything at the last minute. I thank my co-advisor Professor Robert Marano for sharing a passion in software engineering. I thank Professor Cyrus Omar at the University of Michigan, as well as the other members of the FPLab, for providing the technical mentorship for this project despite the unconventional collaboration.

I thank my peers in electrical engineering, for supporting me all this way. I thank the members of the 10B organization, for being my closest friends since the beginning of our Cooper experience. I thank and congratulate Victor Zhang, Derek Lee, and Mark Koszykowski for sharing the experience of the dual-degree program.

Lastly, and most importantly, I thank my family, who have allowed me to spend all my efforts in school unhindered for the past eight years, even through the panic of a pandemic. I thank my parents for the immense support in this process, and my sisters Jessica, Juliet, Josie¹, Sharon¹, and Jane¹ for your emotional support. I thank the numerous family members that have allowed me to reside with them during my stay in the city: my grandma, Uncle Frank, Ben, Uncle John, and Aunt Amy. I hope I can repay your generosity someday. Lastly, I again thank my grandma for being my most lively and loving cheerleader.

¹Not human.

ABSTRACT

Hazel is a live programming environment with typed holes that serves as a reference implementation of the Hazelnut Live dynamic semantics [1] and the Hazelnut static semantics [2], both of which tackle the “gap problem.” This work attempts to further develop the Hazelnut Live dynamic semantics by implementing the environment model of evaluation (as opposed to the current substitution model) and memoizing several evaluation-related operations to improve performance. Additionally, we provide an implementation-level description and a reference implementation of the fill-and-resume (PAR) performance optimization proposed in Hazelnut Live. We produce a metatheory and reference implementation of the proposed changes. Our implementation is benchmarked against the existing Hazel implementation to show that the results match expectations, although there is room for future improvement with the development with memoization. Finally, we discuss some useful theoretical generalizations that result from this work.

TABLE OF CONTENTS

1	Introduction	1
1.1	Problem statement	1
1.2	The contribution of this work	2
1.3	Structural overview	3
2	Programming language principles	4
2.1	Specifications of programming languages	4
2.1.1	Syntax	5
2.1.2	Notation for semantics	6
2.1.3	Static semantics	7
2.1.4	Dynamic semantics	10
2.2	Introduction to functional programming and the λ -calculus	12
2.3	Implementations of programming languages	13
2.3.1	Compiler vs. interpreter implementations	13
2.3.2	The substitution and environment models of evaluation	15
2.4	Approaches to programming interfaces	16
2.4.1	Structure editors	16
2.4.2	Live programming environments and computational notebooks	17
3	An overview of the Hazel programming environment	18
3.1	Motivation for Hazel	18
3.1.1	The gap problem	18
3.1.2	An intuitive introduction to typed expression holes	19
3.1.3	The Hazel interface	20

3.1.4	Implications of Hazel	21
3.2	Introduction to OCaml and Reason syntax	22
3.3	Hazelnut semantics	22
3.3.1	Hazelnut syntax	23
3.3.2	Hazelnut typing and action semantics	23
3.3.3	Hazelnut Live elaboration judgment	24
3.3.4	Hazelnut Live final judgment and dynamic semantics	25
3.3.5	Hole instance numbering	25
3.3.6	High-level overview of fill-and-resume	25
4	Implementing the environment model of evaluation	26
4.1	Hazel-specific implementation	26
4.1.1	Evaluation rules	26
4.1.2	Evaluation of holes	28
4.1.3	Evaluation of recursive functions	28
4.1.4	Type safety	31
4.2	The evaluation boundary and general closures	32
4.2.1	Evaluation of failed pattern matching using generalized closures	33
4.2.2	Generalization of existing hole types	34
4.2.3	Formalizing the evaluation boundary	36
4.2.4	Alternative strategies for evaluation past the evaluation boundary	38
4.2.5	Pattern matching for closures	39
4.3	The postprocessing substitution algorithm ($\uparrow\Downarrow$)	40
4.3.1	Substitution within the evaluation boundary ($\uparrow\Downarrow_{1,1}$)	40
4.3.2	Substitution outside the evaluation boundary ($\uparrow\Downarrow_{1,2}$)	41
4.3.3	Post-processing memoization	43
4.4	Implementation considerations	45

4.4.1	Data structures	45
4.4.2	Additional constraints due to hole closure numbering	46
4.4.3	Storing evaluation results versus internal expressions	46
5	Identifying hole closures by physical environment	47
5.1	Rationale behind hole instances and unique hole closures	47
5.2	The existing hole instance numbering algorithm	49
5.3	Issues with the current implementation	50
5.3.1	Hole instance path versus hole closure parents	50
5.4	Algorithmic concerns and a two-stage approach	52
5.4.1	The hole numbering algorithm	53
5.4.2	Hole closure numbering order	53
5.4.3	Unification with substitution postprocessing	55
5.4.4	Characterizing hole numbering	56
5.5	Fast structural equality checking	57
6	Implementation of fill-and-resume	58
6.1	Motivation	58
6.2	The FAR process	59
6.2.1	Detecting the fill parameters via structural diff	60
6.2.2	“Fill”: pre-processing the evaluation result for re-evaluation	66
6.2.3	“Resume”: Modifications to allow for re-evaluation	68
6.2.4	Post-processing resumed evaluation	70
6.3	Entrypoint to the FAR algorithm	71
6.4	Characterizing FAR	72
6.5	FAR examples	73
6.5.1	Motivating example	73
6.5.2	Introducing and removing static type errors	74

6.5.3	Example requiring recursive evaluation of closure environment	74
6.5.4	Hole fill expression memoization example	76
6.5.5	Noteworthy non-examples	76
6.6	Tracking evaluation state	78
6.6.1	Step counting	79
6.7	Differences from the substitution model	79
7	Evaluation of performance	81
7.1	Evaluation of performance using the environment model	82
7.1.1	A computationally expensive fibonacci program	82
7.1.2	Variations on the fibonacci program	82
7.2	Postprocessing performance	87
7.3	FAR performance	87
7.3.1	A motivating example	87
7.3.2	Decreased performance with FAR	90
8	Discussion of theoretical results	95
8.1	Expected performance differences between evaluating with substitution versus environments	95
8.2	Purity of implementation	96
8.3	Summary of metatheorems	97
8.4	FAR for notebook-style editing	97
8.5	Summary of generalized concepts	99
8.5.1	Generalized closures and the evaluation boundary	99
8.5.2	A generalization of non-empty holes	100
8.5.3	FAR as a generalization of evaluation	100
9	Future work	101
9.1	Simplification of the postprocessing algorithms	101

9.2	Improvements to FAR	101
9.2.1	Finishing the implementation of FAR	101
9.2.2	Memoization for environments during re-evaluation	102
9.2.3	Choosing the edit state to fill from	103
9.2.4	User-configurable FAR	103
9.2.5	UI changes for notebook-style evaluation	104
9.3	Collection of editing statistics	104
9.4	Generalization of memoized methods	105
9.5	Mechanization of metatheorems and rules	105
10	Conclusions and recommendations	106
	References	109
	Appendices	112
A	Primer to the λ -calculus	113
A.1	The untyped λ -calculus	113
A.2	The simply-typed λ -calculus	115
A.3	The gradually-typed λ -calculus	117
B	Reproduced rules from Hazelnut Live	122
B.1	Hazelnut Live dynamic semantics	122
B.1.1	Final judgment	122
B.1.2	Substitution-based evaluation judgment	122
B.2	Hazelnut Live substitution-based FAR	122
C	Selected code samples	126
C.1	Correspondence between theory and code	126
C.2	Relevant code snippets	126

C.2.1	Internal language	127
C.2.2	Numbered environments	130
C.2.3	Evaluation	134
C.2.4	Postprocessing	142
C.2.5	Unique hole closures	153
C.2.6	FAR	160
C.2.7	Evaluation state	168

LIST OF FIGURES

1.1	Screenshot of the Hazel live programming environment	2
2.1	A sample grammar	5
2.2	Notation for an inference rule	6
2.3	Sample typing rules	8
2.4	Simple bidirectional type system	9
2.5	Small-step dynamic semantics for addition	10
2.6	Big-step dynamic semantics for addition	11
2.7	Values in Λ	11
3.1	The Hazel interface, annotated	21
3.2	Hazelnut syntax	23
4.1	Updates to evaluation rules for the environment model of evaluation	27
4.2	Evaluation of fixpoints with the environment model	30
4.3	Evaluation rule for recursion using self-recursive data structures	30
4.4	Comparison of internal expression datatype definitions (in module <code>DHExp</code>) for non-generalized and generalized closures	35
4.5	Revised notation for generalized closure	35
4.6	Unevaluated judgment	37
4.7	Subexpression judgment	38
4.8	Updates to final judgments with generalized closures	40
4.9	Substitution postprocessing	42
5.1	Structure of the result of the program in Listing 4	51

5.2	Numbered hole instances in the result of Listing 4	51
5.3	Hole closure numbering postprocessing semantics	54
5.4	Overall postprocessing judgment	56
6.1	1-step vs. n -step FAR	61
6.2	Structural diffing abbreviated judgments	62
6.3	Form equality judgment	63
6.4	Structural diffing of different expression forms	65
6.5	Structural diffing of non-hole expressions	65
6.6	Structural diffing of hole expressions	66
6.7	Revised evaluation rules for closures	69
6.8	Fill-memoized re-evaluation	69
6.9	Previous action call graph	70
6.10	Current action call graph	71
6.11	FAR simple example	74
6.12	FAR introduce static type error example	75
6.13	FAR remove static type error example	75
6.14	FAR fill hole in hole environment example	75
6.15	FAR hole closure memoization example	77
6.16	FAR infix operator fill	78
7.1	Performance of evaluating $\text{fib}(n)$	85
7.2	Performance of evaluating $\text{fib}(n)$ with extra global variables	86
7.3	Performance of evaluating $\text{fib}(n)$ with an unused branch	86
7.4	Performance of evaluating program in Listing 4	89
7.5	Number of evaluation steps per edit in Table 7.3	93
8.1	Comparison of notebook-style programs in MATLAB and Hazel	98

A.1	Grammar of Λ	114
A.2	Dynamic semantics for Λ	114
A.3	Syntax of Λ_{\rightarrow}	115
A.4	Static semantics of Λ_{\rightarrow}	116
A.5	Dynamic semantics of Λ_{\rightarrow}	116
A.6	Updated static semantics of $\Lambda_{\rightarrow}^?$	118
A.7	Matched arrow judgment	119
A.8	Type consistency judgment	119
A.9	Elaboration in Λ_{\rightarrow}^2	120
A.10	Dynamic semantics of $\Lambda_{\rightarrow}^{(r)}$	121
B.1	Hazelnut Live final judgment	123
B.2	Hazelnut Live evaluation rules	124
B.3	Hazelnut Live substitution-based FAR	125

LIST OF TABLES

1	Common notation for the λ -calculus	xviii
2	The λ -calculi	xviii
3	Hazel evaluation judgments	xviii
4	The gradually-typed λ -calculus	xviii
5	Hazel internal language	xix
6	Hazel postprocessing judgments	xix
7	Fill-and-resume structural diff algorithm	xix
8	Fill-and-resume pre-processing and evaluation	xix
7.1	Time (ms) to compute $\text{fib}(n)$	85
7.2	Performance of program illustrated in Listing 4	88
7.3	A program edit history with an expensive computation	92
7.4	A sample edit history for a simple program	94
C.1	Correspondence between symbols and code	126
C.2	Correspondence between algorithms and code	127

LIST OF LISTINGS

1	Illustrating the problem with postprocessing with recursive closures	31
2	Illustration of hole instances	48
3	Illustration of physical equality for environment memoization	49
4	A Hazel program that generates an exponential (2^N) number of total hole instances	51
5	A sample program with an expensive calculation stored in a hole's environment	59
6	A computationally expensive Hazel program with no holes	83
7	Adding global bindings to the program in Listing 6	84
8	Adding variable substitutions to unused branches to the program in Listing 6	84
9	<code>DHExp.rei</code>	128
10	<code>DHExp.re</code>	130
11	<code>VarBstMap.re</code>	131
12	<code>EvalEnv.rei</code>	132
13	<code>EvalEnv.re</code>	133
14	<code>EvalEnvId.rei</code>	134
15	<code>EvalEnvId.re</code>	134
16	<code>Evaluator.rei</code>	134
17	<code>Evaluator.re</code>	142
18	<code>EvalPostprocess.rei</code>	142
19	<code>EvalPostprocess.re</code>	150
20	<code>Program.re</code>	151
21	<code>Result.rei</code>	152
22	<code>Result.re</code>	153

23	MetaVar.rei	154
24	MetaVar.re	154
25	HoleClosureId.rei	154
26	HoleClosureId.re	154
27	HoleClosure.rei	155
28	HoleClosure.re	155
29	HoleClosureInfo..rei	156
30	HoleClosureInfo..re	158
31	HoleClosureInfo.rei	159
32	HoleClosureInfo.re	159
33	HoleClosureParents.rei	160
34	HoleClosureParents.re	160
35	FillAndResume.rei	161
36	FillAndResume.re	164
37	DiffDHExp.rei	164
38	DiffDHExp.re	168
39	Model.re	168
40	EvalState.rei	169
41	EvalState.re	169
42	EvalStats.rei	170
43	EvalStats.re	170

LIST OF METATHEOREMS

4.1.1 Metatheorem (Preservation)	32
4.1.2 Metatheorem (Progress)	32
4.2.1 Metatheorem (Evaluation boundary)	37
4.2.2 Metatheorem (Singular evaluation boundary)	37
4.3.1 Metatheorem (Substitution postprocessing closures)	43
4.3.2 Metatheorem (Evaluation with environments correctness)	43
4.3.3 Metatheorem (Use of id_σ as an identifier)	44
5.4.1 Metatheorem (Hole numbering postprocessing)	56
6.4.1 Metatheorem (Filling (FAR static correctness))	72
6.4.2 Metatheorem (Commutativity (FAR dynamic correctness))	72
6.4.3 Metatheorem (Fill operation)	73
6.4.4 Metatheorem (Resume operation)	73

LIST OF NOTATIONS

e	(External) expression
τ	Type
Γ	Typing context
Δ	Hole context
$\Gamma \vdash e : \tau$	Type judgment
$\Gamma \vdash e \Rightarrow \tau$	Synthetic type judgment
$\Gamma \vdash e \Leftarrow \tau$	Analytic type judgment
$[e'/x]e$	Substitution of e' for x in e

Table 1: Common notation for the λ -calculus

Λ	Untyped λ -calculus
Λ_{\rightarrow}	Simply-typed λ -calculus
$\Lambda_{\rightarrow}^{\circ}$	Gradually-typed λ -calculus
$\Lambda_{\rightarrow}^{\text{H}}$	λ -calculus with typed holes

Table 2: The λ -calculi

d value	Value
d final	Final
d uneval	Contains unevaluated subexpression
$\sigma \vdash d \Downarrow d'$	Evaluation

Table 3: Hazel evaluation judgments

$*$	Unknown type
$\tau \sim \tau'$	Type consistency
$\tau \blacktriangleright \tau_1 \rightarrow \tau_2$	Matched arrow judgment
$\Gamma \vdash e \rightsquigarrow d : \tau$	Elaboration judgment
$d(\tau \Rightarrow \tau')$	Cast expression

Table 4: The gradually-typed λ -calculus

d	Internal expression
$\lambda x : \tau. d$	λ -abstraction
$\text{fix } f : \tau. d$	Fixpoint
σ	Environment
x, f	Variable
u	Hole number
i	Hole instance or closure number
$\textcolor{purple}{\textcircled{i}}^{ui}$	Empty hole
$\textcolor{red}{(d)}^{ui}$	Non-empty hole
$[\sigma]d$	Generalized closure
$d \subseteq d'$	Subexpression (non-recursive)
$d \in d'$	Subexpression (recurring through environments)

Table 5: Hazel internal language

H	Hole instance/closure information
hid	Hole instance/closure id generation function
p	Hole closure parent
$d \uparrow d'$	Postprocessing
$d \uparrow\!\! \uparrow d'$	Postprocessing (substitution)
$H, p \vdash d \uparrow_i d' \dashv H'$	Postprocessing (hole closure numbering)

Table 6: Hazel postprocessing judgments

$d_1 \sqsupseteq d_2$	No diff (empty diff) between d_1 and d_2
$d_1 \triangleright_d d_2$	Non-fill diff from d_1 to d_2
$d_1 \blacktriangleright_d^u d_2$	Fill diff from d_1 to d_2 with fill parameters u and d
$d_1 \ntriangleleft_d^u d_2$	Some (non-empty) diff from d_1 to d_2
$d_1 \triangleright_d^u d_2$	Any (possibly empty) diff from d_1 to d_2
$d_1 \sim d_2$	Expression form equality

Table 7: Fill-and-resume structural diff algorithm

$\llbracket u/d \rrbracket d' = d''$	Fill-and-resume operation on an expression
$\llbracket u/d \rrbracket \sigma = \sigma'$	Fill-and-resume operation on an environment
$\llbracket \sigma \rrbracket d$	Closure with re-eval flag set
$\textcolor{red}{(d)}_i$	Expression d filled in hole with hole closure number i
ρ	Fill memoization context
$\sigma, \rho \vdash d \Downarrow d' \dashv \rho'$	Fill-memoized evaluation

Table 8: Fill-and-resume pre-processing and evaluation

Chapter 1

Introduction

1.1 Problem statement

Unstructured plaintext editing has remained the dominant mode of programming for decades, but lack of structure complicates the implementation of editor services to aid the programming process. Structural editors force a program to be syntactically well-formed, thus eliminating many meaningless program states that are difficult to analyze. Several structural editors, such as Scratch [4], Landu [5], and mbeddr [6], have been proposed to improve the programming experience and improve editor services, such as the elimination of syntax errors or graphical editing.

Hazel [7] is an experimental structural language definition and implementation that aims to solve the “gap problem”: spatial and temporal holes that temporarily prevent code from being able to be compiled or evaluated. In addition to the use of a structural editor to eliminate syntax errors, Hazel also eliminates static type errors and dynamic runtime errors so that all program states are meaningful and amenable to editor services. The structural editor is defined by the bidirectional edit calculus Hazelnut [1], which governs the structural editor and the static semantics (typing rules) of the language. The dynamic semantics (evaluation semantics) are defined by Hazelnut Live [2].

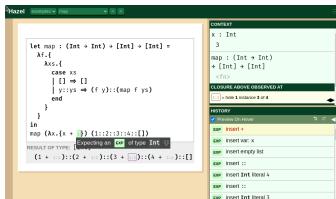


Figure 1.1: A screenshot of the Hazel live programming environment. Screenshot taken of the dev branch demo on 02/06/2022 [3].

Hazel is a relatively new research effort by the University of Michigan's Future of Programming Lab (FPLab), with little effort placed on performance optimizations. This work attempts to achieve several enhancements to Hazelnut Live that will benefit the performance of evaluation and related tasks. Part of the work will be focused on transitioning the evaluation model from using substitution for variable bindings to using environments, with emphasis on evaluation of holes and postprocessing of the evaluation result to match the result from evaluation with substitution. The latter parts of this work will use the environment model of evaluation to improve the memoization of certain tasks specific to Hazel (such as hole closure numbering), and also implement the fill-and-resume performance enhancement described in [2]. The novelty of this work lies in the optimization capacity in the unique design of the Hazel language as a live programming editor with expression holes.

1.2 The contribution of this work

This thesis presents several algorithms designed for Hazel's evaluation. These algorithms are provided using the big-step inference semantics notation introduced in section 2.1.2.

Firstly, we provide the evaluation semantics of the Hazel language using the environment model. We aim to keep the implementation pure, introduce uniquely-numbered environments

(for later use in memoization), and describe the evaluation of holes (which are unique to Hazel). We introduce the concepts of generalized closures, and the evaluation boundary.

Secondly, we describe the postprocessing algorithm, which is mostly memoized by environments and has the two major functions. The first is to convert the result to the equivalent result if substitution was used. The second is to number hole closure instances.

Lastly, we develop the fill-and-resume process, as originally proposed (but not implemented) in [2]. We provide a possible implementation, including an algorithm to detect a valid fill operation and advice on memoizing the resumption operation.

The performance of this work is measured primarily in terms of empirical performance gains (via evaluation-step counting and benchmarking), and discussed with respect to the theoretical performance. Hazelnut [1] and Hazelnut Live [2] mechanize proofs of their work using the Agda interactive proof checker. We do not provide mechanized proofs of our work; instead, we provide a series of metatheorems describing invariants of the Hazel evaluation process, and argue the correctness of these metatheorems by informal reasoning on the provided inference rules. A mechanized proof using Agda is deferred for future work.

1.3 Structural overview

Chapter 2 provides a background on necessary topics in programming language (PL) theory and programming language implementations, in order to frame the understanding for the Hazel live programming environment. Chapter 3 provides an overview of Hazel, in order to frame the work completed for this thesis project. Chapters 4 to 6 describe the primary work completed for this project, as described in Section 1.2. Chapter 7 comprises an empirical performance assessment of the work. Chapter 8 is a discussion of theoretical results. Chapter 9 describes unfinished work and future research directions. Chapter 10 concludes with a summary of findings and future work. The Appendices contain extra inference rules and selected source code snippets.

Chapter 2

Programming language principles

This chapter is intended to provide a primer to the theory of functional programming and programming languages, as relevant to this work on Hazel. The work performed for this thesis is concerned with the dynamic semantics of Hazel.

Section 2.1 is concerned with explaining the notation used throughout this paper to describe formal systems. Section 2.2 is a brief introduction to functional programming and the λ -calculus. A more in-depth explanation of the λ -calculus foundations is given in Appendix A. Section 2.3 provides some detail on different types of programming language implementations relevant to Hazel. In particular, this section sheds some light on the rationale behind switching from an evaluation model based on substitution to an evaluation model based on environments, which forms the basis for a large part of this thesis. Section 2.4 provides an overview of relevant topics in programming language interfaces.

2.1 Specifications of programming languages

Languages are interfaces used for effective communication. Programming languages serve as the interface between programmer and computer. To be able to rigorously work with programming languages, as with any mathematical activity, we need to precisely define their behavior. The definition (or specification) of a programming language is typically given as

```

 $\tau ::= \tau \rightarrow \tau \mid \text{num} \mid \langle \rangle$ 
 $e ::= x \mid \lambda x.e \mid e\ e \mid e + e \mid e : \tau \mid \langle \rangle \mid \langle e \rangle$ 

```

Figure 2.1: A sample grammar

the combination of its *syntax* and *semantics*, which will be discussed below.

Note that the specification of a programming language is orthogonal to its *implementation(s)*; a programming language may have several implementations, which may have differing support for language features and different performance characteristics. Common classifications of programming language implementations are discussed in section 2.3.1.

2.1.1 Syntax

The syntax of a programming language is defined by a grammar. The grammar described in Hazelnut [1] is reproduced in Figure 2.1 as an example.

In this simple grammar, we have two productions: types and expressions. A type may have one of three forms: the `num` type, an arrow (function) type, or the hole type (similar to the `*` type from the GTLC described in Appendix A.3). An expression may be a variable, a λ -abstraction¹, a primitive binary operation, a type ascription, an empty hole, or a non-empty hole. Hole expressions and the hole type are specific to Hazel. Parentheses are not shown in this grammar; they are optional except to specify order of operations.

Parts of this grammar will be revisited when discussing the λ -calculus described in Appendix A.1, and when discussing Hazel's grammar described in Chapter 3. In particular, this Hazelnut grammar is a superset of the grammar of the GTLC described in Appendix A.3, and a subset of the grammar in the implementation of Hazel, which includes additional forms such as `let`, `case`, and pair expressions. Some of these forms will be important cases for our

¹This may have several names depending on the context and programming language, such as: λ -function, function literal, arrow function, anonymous function, or simply function. λ -abstractions are unary by definition – higher-arity functions may be constructed via *function currying*.

p_1	p_2	\dots	p_n	$\xrightarrow{\text{RuleName}}$
q				

Figure 2.2: Notation for an inference rule

study of evaluation.

Due to Hazelnut being a structural edit calculus (as described in section 2.4.1), there is no need to worry about syntax errors. The syntax describes the external language of Hazel, which will be translated into the internal language via the elaboration algorithm prior to evaluation.

2.1.2 Notation for semantics

In formal logic, a standard notation for *rules of inference* is shown in Figure 2.2. p_1, p_2, \dots, p_n are the *antecedents* (alternatively, *premises*) and q is the single *consequent* (alternatively, *conclusion*). Each of p_1, p_2, \dots, p_n, q is a *judgment* (alternatively, *proposition* or *statement*). We may pronounce this rule as “if all of p_1, p_2, \dots, p_n are true, then q must be true.” Note that the antecedent of the rule is the logical conjunction of the antecedents $\bigwedge_{i=1}^n p_i$. The logical disjunction of antecedents $\bigvee_{i=1}^n p_i$ is expressed by writing separate rules with the same consequent. A rule with zero premises is an *axiom*, i.e., the conclusion is vacuously true. We may build up a formal logic system from a set of inference rules. Note that the set of judgments that form the premises of a rule, as well as the set of rules in a formal system, are both unordered; however, any computer program that carries out these judgments must choose some order in which to evaluate the set of antecedents or the order in which to evaluate a set of equally-viable rules.

Each new judgment form will be introduced annotated with the modes of each term. For example, the typing judgment $\Gamma^- \vdash e^- : \tau^+$ indicates that Γ and e are inputs to the judgment and τ is an output of the judgment. If there are no terms with output mode in a judgment, then the ability to logically construct the judgment is its sole (boolean) output.

A derivation (proof) of a judgment is shown by chaining inference rules, such that the final consequent is the statement to be proved. We may visualize a derivation as a tree rooted at the judgment to be proved, and whose children are (recursively) the antecedent judgments; the leaf nodes of this tree must be axioms.

To ensure that the system of inference rules covers the entire semantics of a language, that rules do not conflict, and that rules give the language the desired behavior, we establish *metatheorems*. Metatheorems are intuitive, high-level invariants or properties that describe the behavior of the overall system. We prove the correctness of an implementation by proving that the metatheorems are upheld by the inference rules. In the foundational papers for Hazel's core semantics [1, 2], metatheorems are amply used to justify and verify the correctness of the rules. Agda [8], an interactive proof checker and dependently-typed programming language, is used to mechanize these proofs [9, 10].

2.1.3 Static semantics

The *static semantics* of a programming language describes properties of a program that can be checked prior to program evaluation. Static semantics typically refers to *type checking*. In Hazelnut Live, we have the process of *elaboration* that transforms the *external language* (a program expressed in the syntax of Hazel) to the *internal language* (an intermediate representation more amenable to evaluation), which occurs before evaluation and incorporates the type checking rules. Elaboration and the internal language will be discussed further in section 2.3.1. The type checking and elaboration algorithms form the static semantics of Hazel.

It is formative to provide an overview of type checking. While the static semantics is not very important to the core work in this thesis, a fundamental understanding is key to understanding the motivation and bidirectionally-typed action calculus underlying Hazel, as well as understanding the formulation of gradual typing described in Appendix A.3.

The *typing judgment* $\Gamma^- \vdash e^- : \tau^+$ states that, with respect to the typing context Γ ,

$\frac{}{\Gamma \vdash n : \text{num}}$	TNum
$\frac{}{\Gamma, x : \tau \vdash x : \tau}$	TVar
$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.e) : \tau_1 \rightarrow \tau_2}$	TAnnArr
$\frac{\Gamma \vdash e_2 : \tau_1 \quad \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_1 e_2 : \tau_2}$	TAp

Figure 2.3: Sample typing rules

the expression e is well-typed with type τ . The typing context is a set of variable typing judgments $\{x : \tau\}$. A few sample typing judgments are shown in Figure 2.3.

There are a few noteworthy items here. The syntax $\Gamma, x : \tau$ indicates the typing context Γ extended with the binding $x : \tau$. Thus, when this notation is part of the consequent, it means that we are stating the typing judgment with respect to a different typing context $\Gamma' = \Gamma, x : \tau$. The type of a number is always `num`. The type of a variable may only be determined if its type exists in the typing context (which, according to this limited set of rules, may only be extended during a function application). Lambda expressions can only be typed if they are fully-annotated: i.e., if the argument's type is annotated and the body is also assigned a type. This example typing system is very minimal and not practical for larger systems: every λ -abstraction would have to be typed for the entire expression to be well-typed. Consider even the simple example $(\lambda x.x) 2$, which cannot be typed according to the simple system above due to the unannotated λ -abstraction.

A type system that allows for fewer type annotations, while remaining reasonably simple to formulate and implement, is *bidirectional typing* [11, 12, 13], or *local type inference*. Bidirectional typing involves two typing judgments: the *synthetic type judgment* $\Gamma^- \vdash e^- \Rightarrow \tau^+$ (pronounced “given typing context Γ , expression e synthesizes type τ ”), and the *analytic type judgment* $\Gamma^- \vdash e^- \Leftarrow \tau^-$ (pronounced “given typing context Γ , expression e analyzes against type τ ”). The synthetic type judgment outputs a type (the exact or “narrowest” type of the expression), whereas the analytic type judgment takes a type as an input and “checks” the

$$\begin{array}{c}
 \frac{}{\Gamma \vdash n \Rightarrow \text{num}} \text{TSynNum} \quad \frac{}{\Gamma, x : \tau \vdash x \Rightarrow \tau} \text{TSynVar} \\
 \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash (\lambda x : \tau_1. e) \Rightarrow \tau_1 \rightarrow \tau_2} \text{TSynAnnArr} \quad \frac{\Gamma \vdash e_2 \Rightarrow \tau_1 \quad \Gamma \vdash e_1 \Leftarrow \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_1 e_2 \Rightarrow \tau_2} \text{TSynAp} \\
 \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x. e \Leftarrow \tau_1 \rightarrow \tau_2} \text{TAnaArr} \quad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau} \text{TAnaSubsume}
 \end{array}$$

Figure 2.4: Simple bidirectional type system

expression against that (“wider”) type. With these two judgments, we loosen the antecedent judgments when synthesizing a type. We re-express the above type assignment system into a similar bidirectional type system, shown in Figure 2.4.

Now, we may synthesize the type of $(\lambda x. x)$; 2; the derivation uses all of the rules above. Note the presence of the last rule; *subsumption* states that an expression analyzes against its synthesized type, which should fit the earlier intuition of type synthesis producing the “narrowest” type and type analysis checking against a “wider” type. Subsumption allows us to avoid manually writing type analysis rules for most types.

Algorithmically, bidirectional typing begins by synthesizing the type of the top-level expression; if it successfully synthesizes, then the expression is well-typed. A more complete discussion of bidirectional typing is left to Dunfield [11], who provides an overview of bidirectional typing, or to the formulation of Hazelnut’s bidirectional typing. Hazelnut is at its core a bidirectionally-typed “edit calculus” [1], citing the balance of usability and simplicity of implementation.

The elaboration algorithm is bidirectionally-typed and fairly specific to Hazel and described in section 3.3.4. It is based off of the cast calculus from the GTLC.

More advanced type inference algorithms such as type unification are used in the highly advanced type systems of languages such as Haskell [14], and are out of scope for this work.

$$\begin{array}{c}
 \frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \text{ EPlus}_1\text{-Small} \quad \frac{e_2 \rightarrow e'_2}{\underline{n_1} + e_2 \rightarrow \underline{n_1} + e'_2} \text{ EPlus}_2\text{-Small} \\
 \hline
 \frac{\underline{n_1} + \underline{n_2} \rightarrow \underline{n_1 + n_2}}{} \text{ EPlus}_3\text{-Small}
 \end{array}$$

Figure 2.5: Small-step dynamic semantics for addition

2.1.4 Dynamic semantics

The *dynamic semantics* (alternatively, *evaluation semantics*) of a programming language describes the evaluation process. Evaluation is the algorithmic reduction of an expression to a *value*, an irreducible expression.

The style of rules that are used to define the dynamic semantics of a programming language are called *operational semantics*, because they model the operation of a computer when compiling or evaluating a programming language. There are two major styles of operational semantics.

The first of these styles is *structural operational semantics* as introduced by Plotkin [15] (alternatively, *small-step semantics*). In the small-step semantics, the *evaluation judgment* is $e_1^- \rightarrow e_2^+$, where e_1 and e_2 are expressions in the language.

For example, let us describe the dynamic semantics of an addition operation using a small-step semantics. This is described using the three rules shown in Figure 2.5.

The algorithm carries itself out as follows: while e_1 is reducible, reduce it using some applicable evaluation rule. Once e_1 becomes a value, the first rule is no longer applicable (as e_1 cannot further reduce) and e_2 reduces until it too is a value. Finally, the third rule is applicable, and reduces the expression down to a single number literal. Note that if either e_1 or e_2 do not reduce down to a number literal, then the expression will not evaluate fully; this kind of failure cannot happen in a strongly-typed language due to typing rules.

The second of these styles is *natural operational semantics* as introduced by Kahn [16]

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 + n_2} \text{EPlus-B}$$

Figure 2.6: Big-step dynamic semantics for addition

$$\lambda x.e \text{ value} \rightarrow \text{VLam}$$

Figure 2.7: Values in Λ

(alternatively, *big-step semantics*). In the big-step semantics, the evaluation judgment is $e^- \Downarrow v^+$, where e is an expression in the language, and v value.

To express the evaluation of addition in the big-step semantics, we need only a single rule, shown in Figure 2.6. In this case, the antecedents indicate that the subexpressions must be recursively evaluated, but (as noted earlier) this notably doesn't specify the order of evaluation of the antecedents, unlike the small-step notation.

In the big-step notation, values are distinguished by the judgment $v \Downarrow v$; i.e., values evaluate to themselves. In the small-step notation, there will be no applicable rule to further reduce a value. Following the notation from [1, 2], we can alternatively write this using the equivalent judgment $v^- \text{ value}$. In the stereotypical untyped λ -calculus, the only values are λ -abstractions. We can denote this using the axiom in Figure 2.7.

In Hazel, we have other base types such as integers, floats, and booleans, which also have axiomatic value judgments. For composite data such as pairs or injections (binary sum type constructors), the expression is a value iff its subexpression(s) are values.

The implementation of an evaluator with a program stepper capability (as is commonly found in debugger tools) is more amenable to implementation using a small-step operational semantics, since it precisely details the order of sub-reductions when evaluating an expression. The evaluation semantics of Hazelnut Live are originally described using a small-step

semantics in [2]. To simplify the rules, the concept of an *evaluation context* \mathcal{E} is used to recurse through subexpressions.

The big-step semantics is often simpler because it involves fewer rules, and is more efficient to implement. As a result, the implementation of evaluation in Hazel more closely follows the big-step semantics, and it is the notation used predominantly throughout this work.

2.2 Introduction to functional programming and the λ -calculus

To understand this work on extending Hazel’s dynamic semantics, one must have a satisfactory understanding of Hazel. Understanding Hazel requires some understanding of the *functional programming paradigm*, as Hazel is a stereotypical functional language.

Functional programming [17] is a programming paradigm that is highly involved with function application, function composition, and first-class functions. It is a subclass of the declarative programming paradigm, which is concerned with pure² expression-based computation. Declarative programming is often considered the complement of imperative programming, which may be characterized as programming with mutable state, side effects, or statements. Purely functional programming is a subset of functional programming that deals solely with pure functions; non-pure languages may allow varying degrees of mutable state but typically encourage the use of pure functions.

Functional languages are based on Alonzo Church’s λ calculus [18] as its core evaluation and typing semantics, which provides a minimal foundation for computation. The syntax of functional programming languages is based off the λ calculus. This, along with the lack of mutable state and side effects, allows functional programming to be easily mathematically modeled and reasoned about, making it particularly amenable to proofs about programming languages. This is as opposed to imperative programming, in which the mutable “memory

²“Pure” in the sense of a pure function, i.e., without mutable state or side-effects.

cell” interpretation of variables and side-effects complicates formalizations. A number of programming languages incorporate both functional and imperative language features, such as Hazel’s implementation language OCaml [19]. These languages are classified as multi-paradigm programming languages.

Hazelnut’s core calculus is heavily based on the *gradually-typed λ -calculus* (GTLc) introduced by Siek [20, 21]. This itself is an extension of the simply-typed λ -calculus (STLC), which is an extension of the untyped λ -calculus (ULC), the simplest implementation of Church’s λ -calculus. The STLC, the untyped λ -calculus, and Church’s λ -calculus are standard textbook material in programming language theory [18]. We provide a brief self-contained introduction to these formalizations in Appendix A. This will form the foundational understanding for the study of dynamic semantics in this work, as well as a general understanding of Hazel.

2.3 Implementations of programming languages

In order to run programs in a programming language on a computer, we must have an *implementation* of the language. Hazel is implemented as an interpreted language, whose runtime is transpiled to Javascript so that it may be run as a client-side web application in the browser.

It is important to note that the definition of a language (its syntax and semantics) are largely orthogonal to its implementation. In other words, a programming language does not dictate whether it requires a compiler or interpreter implementation, and languages sometimes have multiple implementations.

2.3.1 Compiler vs. interpreter implementations

There are two general classes of programming language implementations: *interpreters* and *compilers* [22]. Both types of implementations share the function of taking a program as

input, and should be able to produce the same evaluation result (assuming an equal and deterministic machine state, equal inputs, correct implementations, and no exceptional behavior due to differences in resource usage).

A compiler is a programming language implementation that converts the program to some low-level representation that is natively executable on the hardware architecture (e.g., x86-64 assembly for most modern personal computers, or the virtualized JVM architecture) before evaluation. This process typically comprises *lexing* (breaking down into atomic tokens) the program text, *parsing* the lexed tokens into a suitable *intermediate representation* (IR) such as LLVM, performing optimization passes on the intermediate representation, and then generating the target bytecode (such as x86-64 assembly) [22]. The bytecode outputted from the compilation process is used for evaluation. Compiled implementations tend to produce better runtime efficiency, since the compilation steps are performed separate of the evaluation, and because there is little to no runtime overhead.

An interpreter is a programming language implementation that does not compile down to native bytecode, and thus requires an interpreter or *runtime*, which performs the evaluation. Interpreters still require lexing and parsing, and may have any number of optimization stages, but do not generate bytecode for the native machine, instead evaluating the program directly.

The term *elaboration* [23] may be used to describe the process of transforming the *external language* (a well-formed, textual program) into the *internal language*. The internal language may include additional information not present in the external language, such as types generated by type inference or bidirectional typing.

The distinction between compiled and interpreted languages is sometimes ambiguous: some implementations feature just-in-time (JIT) compilation that allow “on-the-fly” compilation (e.g., common implementations of the JVM and CLR [24]), and some implementations may perform the lexing and parsing separately to generate a non-native bytecode representation to be later evaluated by a runtime. A general characterization of compiled vs. interpreted languages is the amount of runtime overhead required by the implementation.

Hazel is a purely interpreted language implementation, since optimizations for speed are not among its main concerns. However, performance is clearly one of the main concerns of this thesis project, but the gains will be algorithmic and use the nature of Hazel's structural editing and hole calculus to benefit performance, rather than changing the fundamental implementation. However, an environment-based evaluation model as described in section 2.3.2 may be helpful in implementing a compiled Hazel implementation.

2.3.2 The substitution and environment models of evaluation

This section describes two methods to evaluate a *variable binding*. In the λ -calculus, variables are bound during function application. They are also bound in Hazel using `let` or `case` expressions.

Evaluation of variable bindings in Hazel is formulated and implemented using the *substitution model*. In this model, when a variable is bound, each instance of the variable in the scope of the binding is immediately substituted by the definiens. This is a simple theoretical model for `let` bindings and function application. This is useful for teaching purposes and simple to formulate because it is stateless.

However, for the purpose of computational efficiency, a model in which bound values are lazily expanded (“looked-up”) in some runtime environment only when needed is desirable. This is called the *environment model*, and generally is more efficient because the runtime does not need to perform an extra substitution pass over subexpressions and because untraversed (unevaluated) branches do not require substituting. Also, the runtime does not need to carry an expression-level IR of the language, due to the fact that the substitution model manipulates expressions, while evaluation does not. This means that the latter is more amenable for compilation, and is how compiled languages tend to be implemented: each frame of the theoretical stack frame is a *de facto* environment frame. While switching from the substitution to environment model is not an improvement in asymptotic efficiency, these effects are useful especially for high-performance and compiled languages.

Note that the use of substitution or environments for the eager and lazy evaluation of variable bindings is orthogonal to strict (applicative-order) or lazy (normal-order) evaluation of function arguments [25]. Laziness for efficiency is a pervasive concept in programming languages.

2.4 Approaches to programming interfaces

The most traditional programming interface is the text source file. In this case, we describe two innovations on plaintext files that are relevant to Hazel's design and use cases.

2.4.1 Structure editors

Structure editors form a class of programming language editors that allow one to directly interface with the *abstract syntax tree* of a programming language via a restricted set of *edit actions*, rather than via the manipulation of unstructured plaintext. An immediate benefit of this is the elimination of the entire class of errors related to syntax.

A major use case for structural editing is for the purpose of programming education. By eliminating syntactic errors, the student may shift their attention towards semantic issues in their code. For example, Carnegie Mellon University developed a series of structural editors (GNOME, MacGnome, and ACSE) targeted at programming education [26]. Scratch is a graphical structural editor targeted at younger students (aged 8-16) developed at MIT [4]. Programming education is one of the main proposed use cases for Hazel, such as its use in Hazel Tutor [27].

Structure editors are not limited to programming education. mbeddr is a structure editor for embedded programming [6], and Landu is a structure editor for a Haskell-like functional programming language [5].

One of the major drawbacks of structural editing is the decrease in usability by restricting the set of edit actions. The degree to which an editor “resists local changes” is a property

known as *viscosity* [28]. Structural and visual editing is expectedly more viscous than unstructured plaintext editing. Reducing the viscosity of structural editing is not a goal of Hazel, but a related project at the University of Michigan, Tylr [29], tackles the problem of editing viscosity and may make its way into future versions of Hazel.

Omar et al. [1] describes several other structure editors and their relation to Hazel, and in particular other structure editors which also attempt to maintain well-typedness or operate on formal definitions of an underlying language.

2.4.2 Live programming environments and computational notebooks

Burckhardt et al. describes live programming environments as providing continuous feedback that narrows the “temporal and perceptive gap between program development and code execution” [30]. A common example of a live programming environment are read-evaluate-print loops (REPLs), which allow line-by-line evaluation of expressions. Computational notebooks form an example of live programming environments.

Computational notebooks, such as in IPython/Jupyter Notebook [31] or MATLAB, is another trend in programming languages that has been popular in scientific applications. They provide much feedback about program’s dynamic state, especially interactively or graphically. Notebook-style editing allows one to intersperse editing and evaluation of a program. Programs may be run in sections (potentially out of order), maintaining state between sections evaluations – this is typically for efficiency reasons. There is a large design space in current computational notebooks, with many possible variations in code evaluation, editing semantics, and displaying notebook outputs [32].

Hazel may be considered a live editor as it attempts to eliminate the feedback gap, by providing static and dynamic feedback throughout the lifetime of then program. The fill-and-resume functionality described in [2] and implemented in this work provide a novel possible implementation of notebook-like partial evaluation.

Chapter 3

An overview of the Hazel programming environment

Hazel is the reference implementation for the Hazelnut bidirectionally-typed action semantics and the Hazelnut Live dynamic semantics, both of which are intended to mitigate the *gap problem*. Hazel is intended to serve as a proof-of-concept of these underlying calculi, but recent editions of the implementation are becoming increasingly practical. The reference implementation is an interpreter written in OCaml and transpiled to Javascript using the `js_of_ocaml` (JSOO) library [33] so that it may be run client-side in the browser. A screenshot of the reference implementation is shown in Figure 1.1 [3]. The source code may be found on GitHub [7]. Hazel may be characterized as a purely functional, statically-typed, bidirectionally-typed, strict-order evaluation, structured editor programming language.

3.1 Motivation for Hazel

3.1.1 The gap problem

Programming editor environments aim to provide feedback to a programmer in the form of editor services such as syntax highlighting or language server protocol (LSP) warnings. Live

programming environments aim to provide continuous static (static type error) and dynamic (run-time type error) feedback in real-time, allowing for rapid prototyping. However, over the course of the lifetime of a program, the program may enter many edit states when it is *meaningless* (ill-formed or ill-typed).

Editor services can only assign static and dynamic meaning to programs that are statically well-typed and free of dynamic type errors. Some may deploy reduced *ad hoc* algorithms for meaningless edit states. This means that over the course of editing, the programmer experiences temporal gaps between moments of complete editor services. This is known as the *gap problem* [30, 2].

3.1.2 An intuitive introduction to typed expression holes

Hazelnut and Hazelnut Live address the gap problem by defining the static and dynamic semantics, respectively, for a small functional programming language extended with typed expression holes. It is built on top of a *structure editor*, which ensures that a program is always well-formed (syntactically correct) by disallowing invalid edit actions. The Hazelnut action semantics for typed holes ensures that a well-formed program is always well-typed. The Hazelnut Live dynamic semantics defines an encapsulated behavior for type errors, such that evaluation continues “around” and captures information about type errors in order to provide dynamic feedback to the programmer.

The Hazelnut Live paper provides the following intuitive understanding of holes.

Empty holes stand for missing expressions or types, and non-empty holes operate as “membranes” around static type inconsistencies (i.e. they internalize the “red underline” that editors commonly display under a type inconsistency).

We have already acknowledged the existence of type holes in dynamically-typed languages and in the $\Lambda^{(\tau)}_*$, in which type holes are represented by the type $*$. This allows unannotated expressions to statically type-check, with the possibility of running into a dynamic type error

at runtime.

Some languages also have the concept of expression holes, which allow a program to be well-typed with missing expressions. In Haskell, for example, the special error value `undefined` always type-checks but will immediately crash the program if it is encountered during evaluation. Haskell also provides the syntax `_u` for typed expression holes, which provides static type information but will not compile¹. Hazelnut Live is the first example of a dynamic semantics that does not consider the evaluation of holes as an exceptional behavior that would crash the program.

In summary, Hazel provides empty type and expression holes, which represent dynamic typing and missing expressions. Nonempty holes are also provided to encapsulate error conditions and provide a well-defined dynamic semantics while providing useful feedback to the user. The dynamic semantics is carefully defined to stop when such indeterminate expressions are encountered, but continue elsewhere (“around” holes or failed casts) if possible.

3.1.3 The Hazel interface

In Figure 3.1 the web interface for the Hazel live environment is shown. The left panel marked (1) is a informational panel showing the list of keyboard shortcuts to perform structured edit actions. Since Hazel is a structured editor, simply typing the program as plaintext will not work; one must use the appropriate shortcuts the construct and edit the program. (2) is the code view. Below the code, a gray box indicates the result of evaluating the expression. The program result updates in real time with every edit action. (3) is the context inspector, which shows information about a hole if a hole is selected. It shows the hole environment, the hole’s typing context (static feedback), the values of the variables in scope (dynamic feedback), followed by the path to the hole and the number of hole instances. In this case, the third hole in the result is selected, in which `x` has value 3. Lastly, (4) shows a history of the edit actions. Clicking on a past edit state will revert the program to that edit state.

¹We may force this to compile using the `-fdefer-type-errors` flag, but then holes will crash when encoun-

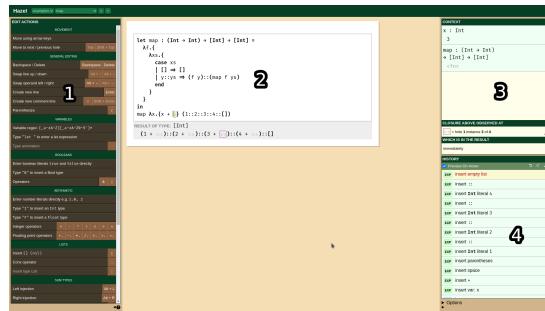


Figure 3.1: The Hazel interface, annotated

3.1.4 Implications of Hazel

The main proposed use case of Hazel is its use in programming education, particularly for teaching functional programming, as it provides much useful feedback to the programmer for error conditions, allowing them to focus instead on semantic errors in their algorithm. This is being explored with the Hazel Tutor project [27].

Another research direction is in its use as a structural and graphical editor. For example, live GUIs [34] are being explored to enhance the editing experience by providing live, compositional, graphical interfaces, in addition to the benefits that Hazel's core calculi provide.

The result of a Hazel evaluation may contain holes. The Hazelnut Live paper [2] suggests the idea of hole-filling: since each hole in the result contains stores its runtime environment, we may “resume” evaluation without restarting evaluation from the beginning if a hole is filled – this property is similar to that of computational notebooks. The problem with typical holes is that they are typically created during evaluation similar to `undefined`.

notebook execution is that it is stateful and running operations out-of-order may cause irreversible state changes that cause irreproducible results. On the other hand, resuming an evaluation with fill-and-resume will always produce the same result as if the program was run ordinarily from start to finish² while avoiding re-evaluation of previous sections.

3.2 Introduction to OCaml and Reason syntax

Previously, we have been introducing concepts using a pseudo-mathematical notation. When describing Hazel and its implementation, it may be useful to use sample code or pseudocode from the implementation to describe various aspects of Hazel.

Hazel is implemented in Reason (alternatively, ReasonML), which is a dialect of OCaml that offers a JavaScript-like syntax. The Reason code used in this report will be limited to function names and types. Module names are denoted **PascalCase**, whereas function and type names are **snake_case**. Conventionally, OCaml modules that export a type export a single type called `t`. As an example, `DHExp.t` refers to the primarily-relevant type from the `DHExp` module, the type that represents internal expressions `d`. On the other hand, `Evaluator.evaluate` refers to the `evaluate` function in the `Evaluator` module. All functions and types will be fully-qualified (prefixed with their module names) for maximum clarity.

3.3 Hazelnut semantics

A high-level overview of the foundational papers on Hazelnut syntax and static semantics and Hazelnut Live elaboration and dynamic semantics is provided here, but a thorough explanation is deferred to the original descriptions [1, 2].

²This is a property known as *commutativity* and described in [2].

$$\begin{aligned}\tau ::= & \tau \rightarrow \tau \mid b \mid \emptyset \\ e ::= & c \mid x \mid \lambda x : \tau. e \mid e.e \mid e : \tau \mid \emptyset \mid \langle e \rangle\end{aligned}$$

Figure 3.2: Hazelnut syntax

3.3.1 Hazelnut syntax

The grammar of Hazelnut’s external language is reproduced in Figure 3.2. This is very similar to Λ_{\rightarrow}^2 . The * type is rewritten as \emptyset and pronounced the “hole type.” An expression form for type ascription is added. Most notably, there is the addition of empty and non-empty expression holes, which are denoted \emptyset and $\langle e \rangle$, respectively.

3.3.2 Hazelnut typing and action semantics

Hazelnut [1] defines a bidirectional typing judgment for the external language. The judgments are very similar to Λ_{\rightarrow}^2 . Unsurprisingly, hole expressions synthesize the hole type, and they analyze against any type. Note that in the case of a non-empty hole, the encapsulated expression must still synthesize a type, i.e., they are well-typed.

Hazelnut defines an action semantics for the structural editor, which describes the behavior of editing and maneuvering around a program. For example, the action semantics automatically add non-empty holes around type errors so that a program is always well-typed. A program’s edit state comprises an external expression with a superimposed cursor. There are four main actions carried out by the user: `move`, `construct`, and `delete`. These actions are described by bidirectionally-typed action judgments that transform a (well-typed) edit state to another (well-typed) edit state. There are a number of metatheorems that enforce desirable properties of action semantics in a structural editor, such as *sensibility* (the result of an action on a well-typed expression is a well-typed expression), *movement* *erase invariance* (movement actions should not change the external expression, but only the position of the cursor), *reachability* (the cursor should be able to move to any valid location

to any other valid location), *constructability* (every valid edit state should be constructable from the initial edit state), *action determinism* (every sequence of edit actions should have only one valid output state), etc. These metatheorems are proved using the Agda theorem proving assistant [9].

The typing and action semantics rules will not be reproduced here, as this work does not concern the static semantics.

3.3.3 Hazelnut Live elaboration judgment

Elaboration is the process of converting an expression from the external language to the internal language. Notably, both the external and internal languages share the same type system. The internal language and the elaboration process is very similar to the cast calculus $\Lambda_{\rightarrow}^{(\tau)}$ and the elaboration process from $\Lambda_{\rightarrow}^{\tau}$.

The elaboration algorithm is also bidirectionally-typed, and thus involves two mutually-recursive judgments: a *synthetic elaboration judgment* $\Gamma^- \vdash e^- \Rightarrow \tau^+ \rightsquigarrow d^+ \dashv \Delta^+$, and an *analytic elaboration judgment* $\Gamma^- \vdash e^- \Leftarrow \tau^- \rightsquigarrow d^+ : \tau'^+ \dashv \Delta^+$. Notably, as a bidirectionally-typed system, the type τ' assigned to holes will be the analyzed type of the expression. Holes will only be assigned the hole type ∅ if the hole appears in a synthetic position. We do not reproduce these rules here, as this work does not concern elaboration.

Δ is the *hole context*, used to store the typing context and actual type of each hole. Each hole (whether in synthetic or analytic position) is recorded in the hole context, and is given the identity mapping as its original environment³.

The elaboration judgment will output a type for the internal expression, which may be different from the type of the external expression. In particular, elaborated holes will produce different types depending on whether they are in synthetic or analytic position.

³This is amended in this work, in which holes will not initially be given an environment because the environment is not substitution-based.

3.3.4 Hazelnut Live final judgment and dynamic semantics

Hazelnut Live introduces a new *d* final judgment for the internal language, used to indicate an irreducible expression in the internal language. Final values subsume values, which now are understood to be deterministic irreducible expressions. With the cast calculus, *boxed values*, i.e., values casted “into” the dynamic (hole) type but not yet casted “out,” are also irreducible. Lastly, *indeterminate values* contain holes which halt evaluation. These are summarized in Appendix B.1.1. As we introduce new internal expression forms, these rules will be modified.

Hazelnut Live defines a small-step semantics for its internal language very similar to that of $\Lambda_{\rightarrow}^{(\cdot)}$. To avoid the rapid proliferation of rules due to the small-step semantics, a notational convenience called the *evaluation context* \mathcal{E} , which recursively evaluates subexpressions. The rules are modified to accomodate indeterminate expressions.

The evaluation rules for Hazelnut Live are reproduced in big-step form in Appendix B.1.2.

3.3.5 Hole instance numbering

Hazelnut Live briefly introduces and motivates *hole instances*, but with no details of its implementation. In Chapter 5, we will motivate hole instances in greater detail, describe the current implementation, and reformulate the problem of hole instance tracking to accomodate environments and memoization.

3.3.6 High-level overview of fill-and-resume

The fill-and-resume optimization (FAR) is motivated and described at a high level in Hazelnut Live. Notably, a dynamic semantics is provided for the operation using substitution, but a full implementation is not provided. Notably, there is no description of how to detect a valid fill operation, or how to cache multiple edit states for *n*-step FAR. FAR is rooted in contextual substitution from contextual modal type theory (CMTT) [35].

Chapter 4

Implementing the environment model of evaluation

4.1 Hazel-specific implementation

The implementation of evaluation in Hazel differs from a typical interpreter implementation of evaluation with environments in three regards. First, we need to account for hole environments. Secondly, environments are uniquely identified by an identifier for memoization (in turn for optimization). Lastly, any closures in the evaluation result should be converted back into λ -abstractions for viewing in the context inspector, so that the result matches the result from evaluation with substitution.

4.1.1 Evaluation rules

The evaluation model threads a run-time environment σ^1 throughout the evaluation process for variable lookups. This replaces the variable substitution pass when evaluating with the substitution model. An environment is conceptually a mapping $\sigma : x \mapsto d$, although it will later be augmented to be more amenable to memoization.

¹The symbol σ was chosen to represent runtime environments, since it was used to represent hole environments in [2]. The relationship between these two environment types will be discussed in section 4.1.2.

$\boxed{d \text{ val}}$ d is a value $\boxed{\sigma[\lambda x : \tau.d \text{ val}]}$ VFunClosure $\boxed{\sigma \vdash d \Downarrow d'}$ d evaluates to d' given environment σ $\frac{}{\sigma \vdash (\lambda x : \tau.d) \Downarrow [\sigma](\lambda x : \tau.d')}$ ELam $\frac{}{\sigma, x \leftarrow d \vdash x \Downarrow d}$ EVar $\frac{\sigma \vdash d_1 \Downarrow [\sigma']\lambda x : \tau.d'_1 \quad \sigma \vdash d_2 \Downarrow d'_2 \quad \sigma', x \leftarrow d'_2 \vdash d'_1 \Downarrow d}{\sigma \vdash d_1 d_2 \Downarrow d} \text{ EAp}$ $\frac{}{\sigma \vdash \emptyset^u \Downarrow \emptyset_\sigma^u} \text{ EvalB-EHole} \qquad \frac{\sigma \vdash d \Downarrow d'}{\sigma \vdash (d)_\sigma^u \Downarrow (d')_\sigma^u} \text{ EvalB-NEHole}$

Figure 4.1: Updates to evaluation rules for the environment model of evaluation

The evaluation rules for evaluation using substitution are described in section 3.3.4 and shown in Appendix B. We present the updates to the rules necessary for evaluation with environments in Figure 4.1. Many of the rules are unchanged and are not repeated, especially the evaluation of casts (EFinal, EApCast, ECastId, ECastSucceed, ECastFail, EGround, EExpand). We also note a change to the set of value judgments: function closures $[\sigma]\lambda x : \tau.d$ are now considered values, and functions $\lambda x : \tau.d$ are not.

As expected, a λ -abstraction binds its lexical environment, forming a function closure (ELam). Variables are not eagerly substituted, but rather looked up in the environment when encountered (EVar). On function application, the expression in function position is now expected to evaluate to a function closure. The closure's environment is extended with the binding of the expression in argument position and the body of the function is evaluated (EAp).

There are many additional forms added to Hazel on top of the base Hazelnut grammar,

such as pairs, `let` expressions, and `case` expressions. The extension of the core rules should be straightforward and these extra expression forms will not be described here. There are two exceptions: a description of recursive λ -abstractions (the fixpoint form) is described in section 4.1.3, and the issue of failed pattern-matching in `let` and `case` expressions is described in section 4.2.1.

4.1.2 Evaluation of holes

In the substitution model, there is no evaluation rule for empty holes; they are final. For non-empty holes, evaluation simply recurses into the subexpression. Notably, the hole environment is not evaluated or updated when the hole is reached; rather, it is filled in by an eager substitution pass when a variable binding is evaluated.

In the environment model, we do not have eager substitution passes. Thus, we update the environment when evaluation reaches a hole by setting the hole environment to the current evaluation (lexical) environment. To facilitate this, we originally give holes no environment (denoted \emptyset^* and $(\text{id})^*$), rather than the identity environment $\text{id}(\Gamma)$. The latter is necessary for substitution, but is not needed anymore.

Note that in the updated interpretation, free variables are excluded from a hole environment. In the substitution case, free variables exist in the environment as the identity substitution $x \leftarrow x$.

We are currently using the subscript notation for hole environments from Hazelnut Live. When discussing the evaluation boundary in section 4.2, we will see that it will be more convenient separate the environment from a hole using the notation for generalized closures.

4.1.3 Evaluation of recursive functions

To handle recursion in a strongly-typed language, we require self-reference. Hazel uses the fixpoint operator from System PCF. The static and dynamic semantics of the fixpoint operator are described in Appendix A.2. In Hazel, the fixpoint does not exist in the external

language, but is inserted automatically by the elaboration process when a λ -abstraction is bound to a variable².

We wish to introduce self-reference when evaluating with environments. Perhaps the simplest way is to use memory references (pointers), in which environments may recursively refer to themselves. We eschew this solution because we wish to keep the purity of the Hazel implementation. We discuss purity of implementation in section 8.2.

We explore two pure solutions. The first is to eliminate the fixpoint by using recursive data structures in OCaml, which simplifies evaluation but creates slight differences in the postprocessed result. The latter is to adapt the fixpoint to the environment model of evaluation. Either method is viable; however, the latter is chosen because it is closer to the original implementation and we do not have the postprocessing issue.

The updated evaluation rules for fixpoint evaluation are shown in Figure 4.2. The evaluation of a fixpoint performs a side-effect on the body expression, which is expected to evaluate to a function closure. Namely, we add a self-reference to the closure environment (EFix). When a variable that stores a fixpoint is looked up, then the fixpoint is unwound (EUnwind). If a variable is not a fixpoint, the regular EVar rule applies. Note that it is possible to always evaluate the variable binding, and thus eliminate the EUnwind rule, but this would be inefficient.

This understanding is consistent with the understanding of fixpoints when using substitution. Fixpoints are unwound when they are encountered, but when evaluating with environments this may occur during a variable lookup.

We may avoid the fixpoint form by using mutually-recursive data structures, so that a closure may contain an environment which contains itself as a binding. This is easy to implement in a language with pointers or mutable references. Mutually-recursive data in

²The current implementation of Hazel only allows for recursion in a limited number of cases. The elaboration process only inserts a fixpoint operator for a type-annotated λ -abstraction. This does not allow for mutual recursion, which could be implemented with fixpoint operators applied to pairs of functions. This may change in future versions of Hazel.

$$\begin{array}{c}
 \frac{\sigma \vdash d \Downarrow [\sigma']d'}{\sigma \vdash \text{fix } f : \tau.d \Downarrow [\sigma, f \leftarrow \text{fix } f : \tau.[\sigma']d']d} \text{ EFix} \\
 \frac{d \neq \text{fix } f : \tau.d'}{\sigma, x \leftarrow d \vdash x \Downarrow d} \text{ EVar} \quad \frac{\sigma \vdash \text{fix } f : \tau.d \Downarrow d'}{\sigma, x \leftarrow \text{fix } f : \tau.d \vdash x \Downarrow d'} \text{ EUwind}
 \end{array}$$

Figure 4.2: Evaluation of fixpoints with the environment model

$$\frac{\sigma' = \sigma, f \leftarrow d'_1 \quad d'_1 = [\sigma']\lambda x.d_1 \quad \sigma' \vdash d_2 \Downarrow d}{\sigma \vdash \text{let } f = \lambda x.d_1 \text{ in } d_2 \Downarrow d} \text{ ERecClosure}$$

Figure 4.3: Evaluation rule for recursion using self-recursive data structures

OCaml is somewhat tricky in the general case, as it requires statically-constructive forms³. In the more general case of mutual recursion, this would likely make implementation very tricky, and it would be more practical to use impure `refs` to achieve self-reference. However, for the simple case of a non-mutually-recursive function, we may statically construct the mutual recursion using the rule shown in Figure 4.3.

Using the recursive environment in closures simplifies evaluation and may lead to a slight uptick in performance, due to the elimination of unwinding steps for fixpoints. However, it complicates the display of recursive functions in the context inspector and structural equality checking, due to infinite recursion. The first problem is re-introducing the `FixF` form during postprocessing (section 4.2) by detecting recursive environments and converting them to `FixF` expressions. The second problem is solved by the fast equality checker for memoized environments described in section 5.5, which is useful even for non-recursive environments. We may also say that using recursive data structures without mutable `refs` is limited by the language limitations, necessitating workarounds even for the simply-recursive case, and

³§10.1: Recursive definitions of values of the OCaml reference describes this in greater detail. Simply put, this prevents recursive variables from being defined as arguments to functions, instead only allowing recursive forms to be arguments to data constructors.

```
let f = λ x . { ⊥1 } in
f f
```

Listing 1: Illustrating the problem with postprocessing with recursive closures

potentially much more complicated workarounds for the mutual recursion case.

There is a nuance that may cause the postprocessed⁴ result to slightly differ from that using the fixpoint form. To illustrate this, consider the simple program in Listing 1. The result will be a closure of hole 1 with the identifiers **x** and **f** in scope. When evaluating using the fixpoint expression, the binding for **f** will be the expression $(\text{fix } f.[\emptyset] \lambda x. \perp^1)$, and the binding for **x** is $((f \leftarrow \text{fix } f.[\emptyset] \lambda x. \perp^1) \lambda x. \perp^1)$. **f** is bound to the closure in the EFix rule, and **x** is bound during EA_p to the evaluated value of **f**.

However, when evaluating with a recursive data structure, both **x** and **f** refer to the same value $d = ((f \leftarrow d) \lambda x. \perp^1)$. It is impossible to discern the two and decide where to begin the “start of the recursion,” i.e., to determine that **f** should be a fixpoint expression and **x** should be a λ -abstraction, at least without significant additional extra effort. Thus to remove the recursion, we may arbitrarily decide that the outermost recursive form should be a λ -abstraction and set the recursive binding in its environment to be a fixpoint expression, which will successfully remove the recursion but mistakenly change some expressions that would be fixpoint forms to λ -expressions. This distinction may not be critical, but it will at least confuse the user. This justifies our use of the fixpoint form for evaluating recursive functions.

4.1.4 Type safety

Type safety of a dynamic semantics ensures that it coheres with the static semantics. In other words, it is a useful check for the correctness of a dynamic semantics. Following the style in ??, we establish type safety via two safety properties: (type) *preservation* and *progress*.

⁴Postprocessing will be discussed in section 4.3.

Proofs are provided for Hazelnut Live using the substitution model. Here we loosely motivate that the switch to using environments for binders maintains these properties.

(Type) preservation (Metatheorem 4.1.1) states that the type of an expression is preserved by evaluation. The proof of this should not differ much. Closures exist now as a wrapper but do not change the type of the underlying expression. Variables are looked up in an environment rather than substituted during binding time, but that does not change the type of the expression. Thus we expect the type of the evaluated result to be the same.

Progress (Metatheorem 4.1.2) states that each expression evaluates to a final expression, i.e. that no expression forms remain unevaluated. This uses the final judgment described in Appendix B.1.1, as well as our modifications described in Figure 4.8. The main changes lie in the evaluation of closures, including function and hole closures. We will discuss the evaluation boundary and how any expression that is unevaluated will be contained inside a closure (Metatheorem 4.2.1).

Metatheorem 4.1.1 (Preservation). *If $\Delta; \emptyset \vdash d : \tau$ and $d \Downarrow d'$, then $\Delta; \emptyset \vdash d' : \tau$.*

Metatheorem 4.1.2 (Progress). *If $\Delta; \emptyset \vdash d : \tau$ then $\exists d'$ such that $d \Downarrow d'$ and d final.*

4.2 The evaluation boundary and general closures

Evaluation with the environment model lazily substitutes variables. Evaluation steps that require the environment (e.g., evaluation of holes and variables) are only performed when evaluation reaches the expression of interest. Evaluation with the substitution model eagerly substitutes using a separate substitution pass. In the evaluation result or in the Hazel context inspector, the user may examine expressions in the internal language. The user expects to see fully substituted values, and closures should not appear directly to the user. For example, free variables in a function body should show the captured expression.

In other words, any unevaluated expression must be “caught up” to the substituted equivalent after evaluation. This requires that the environment be stored alongside the unevalu-

ated expression, and that a postprocessing step should be taken to perform the substitution and discard the stored environment. Note that this is essentially performing a substitution pass after evaluation, but is preferred over substitution during evaluation because it is only performed on the evaluation result (rather than all the intermediate expressions during evaluation). We call this *substitution postprocessing*, and will be discussed in section 4.3.

We first define the *evaluation boundary* to be the conceptual distinction between expressions for which evaluation has reached (“inside” the boundary), and for those that remain unevaluated (“outside” the boundary). This definition will be useful for describing the postprocessing algorithm, closures, and fill-and-resume.

4.2.1 Evaluation of failed pattern matching using generalized closures

There are two cases where an expression in the evaluation result may lie outside the evaluation boundary⁵. The first is in the body of a λ -abstraction. A λ -abstraction evaluates to a closure, and thus captures with it the lexical environment in which the function was defined. The second case is that of an unmatched `let` or `case` expression (in which the scrutinee matches none of the rules), for which the body expression(s) will remain unevaluated in the result without an associated environment⁶. Pattern-matching is not part of Hazelnut Live or in this paper because pattern-matching is not a primary concern of either of these works. However, it is a practical concern with Hazel that arises from the introduction of evaluation with environments.

We solve this by introducing (lexical) *generalized closures*, the product of an arbitrary expression and its lexical environment. Traditionally, the term “closure” refers to *function closures*, which are the product of a λ -abstraction with its lexical environment. Hazelnut Live introduces *hole closures*, which are the product of hole environments with their lexical

⁵A third case will appear in Chapter 6 when we discuss the fill-and-resume optimization.

⁶There is a third place where pattern-matching may fail: the pattern of an applied λ -abstraction may not match its argument. However, this is not an issue since functions are already captured in a closure.

environments, and are fundamental to the Hazel live environment. Hole environments allow a user to inspect a hole's environment in the context inspector, and enable the fill-and-resume optimization described in Chapter 6. We propose generalizing the term “closures” to the definition stated above. Conceptually, all generalized closures represent a “stopped and/or resumable” evaluation using the environment model, as well as the state (the environment) that may be used to resume the evaluation. Similar to the evaluation of function closures, closures are final (boxed) values and evaluate to themselves.

The application of generalized closures to the problem of unevaluated `let` or `case` bodies is straightforward: if there is a failed pattern match, wrap the entire expression in a (generalized) closure with the current lexical environment. Then, the postprocessing can successfully perform the substitution.

4.2.2 Generalization of existing hole types

Consider the abbreviated definition of the internal expression variant type in Figure 4.4. In Figure 4.4a the previous implementation is shown (when evaluating using the substitution model), augmented with a type for function closures. In this version, each expression variant that requires an environment has the environment hardcoded into the variant. In Figure 4.4b the proposed version with generalized closures is shown. The `Lam`, `Let`, and `Case` variants are unchanged. Importantly, the environments are removed from the hole types and a new generalized `Closure` variant is introduced. In this model, a hole, λ -abstraction, unmatched `let`, or unmatched `case` expression is wrapped in the `Closure` variant when evaluated.

The notation used to express a function closure may be extended to all generalized closure types. In particular, the environment for a hole changes from the initial notation used in Hazelnut Live to a notation similar to function closures, shown in Figure 4.5.

This implementation of closures is an improvement in three ways. Firstly, it simplifies the variant types by factoring out the environment, separating the “core” expression from the environment coupled with it. Secondly, it allows for a more intuitive understanding of

<pre>type t = (* Hole types *) EmptyHole(u, i, σ) NonEmptyHole(u, i, σ, d) Keyword(u, i, σ, ...) InvalidText(u, i, σ, ...) FreeVar(u, i, σ, ...) InconsistentBranches(u, i, σ, ...) (* λ expressions and closures *) Lam(x, τ, d) FnClosure(σ, x, τ, d) (* ... *) ;</pre>	<pre>type t = (* Hole types *) EmptyHole(u, i) NonEmptyHole(u, i, d) Keyword(u, i, ...) InvalidText(u, i, ...) FreeVar(u, i, ...) InconsistentBranches(u, i, ...) (* λ expressions and closures *) Lam(x, τ, d) (* Generalized closure *) Closure(σ, d) (* ... *) ;</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(a) Non-generalized closures

(b) Generalized closures

Figure 4.4: Comparison of internal expression datatype definitions (in module DHExp) for non-generalized and generalized closures.

$[\sigma]\lambda x.d$	(function closure)
$[\sigma](\text{d})^u$	(hole closure)
$[\sigma](\text{let } x = d_1 \text{ in } d_2)$	(closure around let)
$[\sigma](\text{case } x \text{ of rules})$	(closure around case)

Figure 4.5: Revised notation for generalized closure

holes in the environment model of evaluation. This solves the question of what environment to initialize a hole with when it is created during the elaboration phase: a hole is simply initialized without a hole environment, much as a function closure is initially without an environment (a plain syntactical λ abstraction). This removes the need for the awkward notation $(\text{hole})^*$ introduced earlier to indicate a hole that has not yet been assigned an environment. Lastly, generalized closures play an important role in the fill-and-resume operation, in which (unevaluated) closures can contain arbitrary subexpressions and allow “resuming” evaluation in the stored environment.

Note that while the generalized closures for the body expressions of λ abstractions, unmatched `let` expressions, and unmatched `case` expressions represent expressions outside of the evaluation boundary, the expressions within non-empty holes (which also are bound to a hole closure) lie within the evaluation boundary. This shows the two goals that generalized closures achieve; to encapsulate a stopped expression (which is used during postprocessing to perform substitution), and to encapsulate an expression to be filled for the fill-and-resume operation.

4.2.3 Formalizing the evaluation boundary

We may characterize the evaluation boundary with two theorems on the evaluated result. First, we need to define three auxiliary judgments.

The $d \vdash \text{uneval}$ unevaluated judgment shown in Figure 4.6 indicates that an expression form directly contains an unevaluated expression, or may contain an unevaluated expression at some point in the future. In Hazelnut Live, function bodies are the only unevaluated expressions. We note also that hole expressions are also considered `uneval`; this is because if they are filled in fill-and-resume, they will contain an unevaluated expression. If we consider Hazel’s more complete syntax, then additional axioms should be added for unmatched `let` and `case` statements. UENotFinal provides an intuitive characteristic of the `uneval` judgment: unevaluated expressions are not final (evaluated). It is trivial to prove UENotFinal

$\boxed{d \text{ uneval}}$ d contains an unevaluated subexpression not in a closure
$\frac{}{\lambda x : \tau. d \text{ uneval}} \text{UELam}$
$\frac{\langle \rangle^u \text{ uneval}}{\langle \rangle^u \text{ uneval}} \text{UEEHole}$

$$\frac{d \text{ uneval}}{d \text{ not final}} \text{UENotFinal}$$

Figure 4.6: Unevaluated judgment

from the three axioms and from the `final` judgment.

Two subexpression judgments are shown in Figure 4.7. $d^- \subseteq d'^-$ indicates that a d is a subexpression of d' , not recursing into hole environments. $d \in d'$ also indicates a subexpression, but allows recursing into closure environments. This distinction is important because all environments lie inside the evaluation boundary.

Metatheorem 4.2.1 (Evaluation boundary). *If $\emptyset \vdash d \Downarrow d_1$ and $d_2 \in d_1$ and $d_2 \text{ uneval}$, then $d_2 \subseteq [\sigma]d_3 \in d_1$.*

Metatheorem 4.2.1 states that all unevaluated subexpressions lie within a closure in the evaluated result. This theorem ensures that the postprocessing substitution and fill-and-resume will have the necessary information to succeed. The justification is straightforward by switching on the evaluation rules. The evaluation rules for λ -abstractions and hole expressions wrap the expression in a closure.

Metatheorem 4.2.2 (Singular evaluation boundary). *If $\emptyset \vdash d \Downarrow d_1$ and $d_2 \in d_1$ and $[\sigma]d_2 \in d_1$ and $[\sigma]d_2 \subset [\sigma']d_3$, then $d_3 = \langle d_2 \rangle^u$.*

Metatheorem 4.2.2 states that there are no nested closures in an expression (not recursing into closure environments). This means that unevaluated expression are always separated from evaluated expressions by a single closure, justifying the term “evaluation boundary” rather than “evaluation boundaries.” The only case when closures may be nested are in the case of non-empty holes, in which the hole expression actually lies within the evaluation

$\boxed{d \subseteq d'} d \text{ is a subexpression of } d'$
$\frac{d \subseteq d}{d \subseteq d} \text{ SEId}$
$\frac{d \subseteq d' \quad d \subseteq \lambda x : \tau, d'}{d \subseteq \lambda x : \tau, d'} \text{ SELam}$
$\frac{d \subseteq d_1 \quad d \subseteq d_2}{d \subseteq d_1 d_2} \text{ SEAp}_1$
$\frac{d \subseteq d_1 \quad d \subseteq d_2}{d \subseteq d_1 d_2} \text{ SEAp}_2$
$\frac{d \subseteq d'}{d \subseteq (\textcolor{violet}{d'})^u} \text{ SENEHole}$
$\frac{d \subseteq d'}{d \subseteq [\sigma]d} \text{ SEClosure}$
$\boxed{d \in d'} d \text{ exists in } d'$
$\frac{d \subseteq d'}{d \in d'} \text{ SEDirect}$
$\frac{d \in \sigma}{d \in [\sigma]d} \text{ SEEEnv}$
$\boxed{d \in \sigma} d \text{ exists in } \sigma$
$\frac{d \in \sigma}{d \in \sigma, x \leftarrow d} \text{ SEEEnv}_1$
$\frac{d \in d'}{d \in \sigma, x \leftarrow d} \text{ SEEEnv}_2$

Figure 4.7: Subexpression judgment

boundary⁷. We justify this theorem in the same manner by using induction on the evaluation rules, keeping track of nested closures and recognizing that the outer closure(s) must be around non-empty holes.

4.2.4 Alternative strategies for evaluation past the evaluation boundary

Without generalized closures, unevaluated expressions (body expressions of λ -abstractions, unmatched `let` expressions, and unmatched `case` expressions) may be reached by ordinary evaluation which is slightly in that a failed lookup (due to in-scope but yet-unbound variables) will leave the variable unchanged⁸. However, this eager evaluation is essentially the same as substitution, and is expensive to do during evaluation. Also, while this specula-

⁷We similarly need to consider the scrutinee of unmatched `let` and `case` statements, which lies within the evaluation boundary.

⁸Ordinarily, a lookup on a `BoundVar` (a variable which is in scope) should never fail during evaluation, and thus throws an exception during evaluation.

tive execution would be reasonable for `let` expressions, it would be highly undesirable for `case` expression, where it is easy to imagine an example where speculative execution of all branches leads to infinite recursion.

Another way to eliminate the case of unmatched expressions is to introduce an exhaustiveness checker to Hazel; then, we can guarantee (at run-time) that a pattern will never fail to match. This would also require changing the semantics of pattern holes, which always fail to match; the behavior may be changed so that pattern holes always match, but do not introduce new bindings. Since the focus of this work is not on patterns, these ideas were not explored and are left for future work in the Hazel project.

4.2.5 Pattern matching for closures

Pattern matching is not the primary focus of this work, but it warrants a brief discussion here. Since we introduce a new `DHExp.t` variant, we also need to implement all the methods that switch on a `DHExp.t`, such as pattern matching.

Pattern matching is implemented in the function `Evaluator.matches`, which has type `(DHPat.t, DHExp.t) => Evaluator.match_result`. If pattern matching succeeds, then an environment containing the matched binding(s) will be returned. Otherwise, pattern matching may be indeterminate (if either the pattern or bound expression is indeterminate), or it may fail. Note that the expression passed to `Evaluator.matches` is already evaluated.

Closures are a unique variant of `DHExp.t` in that they are a container type, whose contained expression determines its behavior during pattern matching. An evaluated closure⁹ may only contain one of four types of expressions: λ -abstractions, holes, unmatched `let` expressions, or unmatched `case` expressions. The former is a boxed value and should match against variables only, and otherwise fail. The latter three are indeterminate and should match against variables and return an indeterminate match otherwise.

Another way to understand this behavior is to consider the updated final judgment once

⁹An evaluated closure is one for which the `re_eval` flag introduced in section 6.2.2 is false. Thus far, all closures we have encountered are evaluated.

$\boxed{[\sigma]\lambda x : \tau.d \text{ val}}$	VFunClosure	$\frac{d \neq \lambda x : \tau.d}{[\sigma]d \text{ indet}}$	IClosure	$\boxed{[\sigma]d \text{ final}}$	FClosure
--------------------------------------------------	-------------	-------------------------------------------------------------	----------	-----------------------------------	----------

Figure 4.8: Updates to final judgments with generalized closures

closures have been taken into account. These updated final judgments are shown in Figure 4.8. Function closures are values (VFunClosure). Other general closures are indeterminate (IClosure). The IClosure rule subsumes the old IEHole and INEHole rules in Appendix B.1.1. FClosure states that all closures are final expressions¹⁰. The derivation of FClosure from VFunClosure and IClosure is trivial.

4.3 The postprocessing substitution algorithm ($\uparrow_{[]}^{\cdot}$)

The substitution postprocessing process aims to perform substitution on expressions that lie outside the evaluation boundary in the evaluation result (an internal expression). The algorithm works in two stages: first inside the evaluation boundary, and then proceeding outside the boundary when a closure is encountered.

The symbol chosen to denote postprocessing is $\uparrow_{[]}^{\cdot}$ ¹¹. The two stages of this algorithm will be denoted $\uparrow_{[],1}^{\cdot}$ and $\uparrow_{[],2}^{\cdot}$, respectively.

4.3.1 Substitution within the evaluation boundary ($\uparrow_{[],1}^{\cdot}$)

When inside the evaluation boundary, all (bound) variables have been looked up and all hole environments assigned, so we do not need to perform any substitution. The main point of this step is to recurse through the expression until a closure is found, at which point we enter the second stage and perform substitution.

¹⁰See the previous comment. Re-evaluatable closures will no longer be final.

¹¹The choice of symbol is somewhat arbitrary, but we may read it as “reverting” some expressions generated by and useful for evaluation (i.e., closures) to a more context-inspector-friendly form, which is in some sense the opposite of evaluation (\cdot). The bracket subscript indicates that this post-processing step is intended to remove closure expressions.

For expressions without subexpressions, the expression is returned unchanged; there is nothing to do. For other non-closure expression types, $\hat{\eta}_{[],1}$ recurses through any subexpressions.

For closure types, we first need to recursively apply $\hat{\eta}_{[],1}$ to all bindings in the closure environment. For non-empty holes, the body is inside the evaluation boundary and thus $\hat{\eta}_{[],1}$ is applied. For other expressions inside a closure, the body expression is outside the evaluation boundary, and thus $\hat{\eta}_{[],2}$ is applied to the body expression, using the closure environment. The closure is then removed.

A λ -abstraction, `let` expression, `case` expression, hole outside of a closure, or a bound variable that has not been looked up, will never exist outside of a closure within the evaluation boundary, so these cases need not be handled.

Note that in the implementation with recursive data structures used to represent environments as described in section 4.1.3, an additional step must be taken before recursing into function closures. Recursive function bindings must be detected and converted to `FixF` expressions to prevent infinite recursion.

4.3.2 Substitution outside the evaluation boundary ($\hat{\eta}_{[],2}$)

When outside the evaluation boundary (and inside a closure), we need to substitute bound variables¹² and assign an environment to holes.

Bound variables are looked up in the environment; this lookup may fail if the variable does not exist in the environment, in which case the variable is left unchanged. For other primary expressions, the expression is left unchanged. When a hole is encountered, it is assigned the closure environment¹³. A closure will never exist outside the evaluation boundary in the evaluation result (by Metatheorem 4.2.2).

¹²The wording is a little tricky here, since there are the `BoundVar` and `FreeVar` internal expression variants, which refer to variables which are in scope or not in scope. However, we may only substitute variables which are in-scope (`BoundVar`) and bound; some instances may not yet be bound.

¹³There is nothing to do at this point for hole closures. The hole closure numbering step will assign a closure identifier to the hole as described in the second postprocessing algorithm in section 5.4.

$d \hat{\uparrow}_\square d'$	d is substitutes to d' inside the evaluation boundary
$c \hat{\uparrow}_\square c$	$\text{PPI}_\square \text{Const}$
$d_1 \hat{\uparrow}_\square d'_1 \quad d_2 \hat{\uparrow}_\square d'_2$	$d_1 d_2 \hat{\uparrow}_\square d'_1 d'_2 \text{ PPI}_\square \text{Ap}$
$\sigma \hat{\uparrow}_\square \sigma' \quad \sigma' \vdash d \hat{\uparrow}_\square d'$	$\frac{\sigma \hat{\uparrow}_\square \sigma' \quad \sigma' \vdash d \hat{\uparrow}_\square d'}{[\sigma]d \hat{\uparrow}_\square d'} \text{ PPI}_\square \text{Closure}$
$\sigma \hat{\uparrow}_\square \sigma'$	σ substitutes to σ' (outside the evaluation boundary)
$\emptyset \hat{\uparrow}_\square \emptyset$	$\text{PPI}_\square \text{EnvNull}$
$\sigma, x \leftarrow d \hat{\uparrow}_\square \sigma', x \leftarrow d'$	$\frac{\sigma \hat{\uparrow}_\square \sigma' \quad d \hat{\uparrow}_\square d'}{\sigma, x \leftarrow d \hat{\uparrow}_\square \sigma', x \leftarrow d'} \text{ PPI}_\square \text{Env}$
$\sigma \vdash d \hat{\uparrow}_\square d'$	d substitutes to d' outside the evaluation boundary
$c \hat{\uparrow}_\square c$	$\text{PPO}_\square \text{Const}$
$x \notin \sigma$	$\sigma, x \leftarrow d \vdash x \hat{\uparrow}_\square d \text{ PPO}_\square \text{BoundVar}$
$\sigma, x \hat{\uparrow}_\square x$	$\text{PPO}_\square \text{UnboundVar}$
$\sigma \vdash d \hat{\uparrow}_\square d'$	$\frac{\sigma \vdash d \hat{\uparrow}_\square d'}{\sigma \vdash \lambda x. d \hat{\uparrow}_\square \lambda x. d} \text{ PPO}_\square \text{Lam}$
$\sigma \vdash d_1 \hat{\uparrow}_\square d'_1 \quad \sigma \vdash d_2 \hat{\uparrow}_\square d'_2$	$\text{PPO}_\square \text{Ap}$
$\sigma \vdash d_1(d_2) \hat{\uparrow}_\square d'_1(d'_2)$	$\frac{\sigma \vdash d_1 \hat{\uparrow}_\square d'_1 \quad \sigma \vdash d_2 \hat{\uparrow}_\square d'_2}{\sigma \vdash d_1(d_2) \hat{\uparrow}_\square d'_1(d'_2)} \text{ PPO}_\square \text{Ap}$
$\sigma \vdash d \hat{\uparrow}_\square d'$	$\text{PPO}_\square \text{EHole}$
$\sigma \vdash (d^u)^u \hat{\uparrow}_\square [\sigma](d^v)^v$	$\frac{\sigma \vdash d \hat{\uparrow}_\square d'}{\sigma \vdash (d^u)^u \hat{\uparrow}_\square [\sigma](d^v)^v} \text{ PPO}_\square \text{NEHole}$

Figure 4.9: Substitution postprocessing

Note that the $\hat{\uparrow}_{\square,1}$ algorithm only takes an internal expression d as its input, whereas the $\hat{\uparrow}_{\square,2}$ algorithm takes an internal expression d and a (closure) environment σ as inputs.

We may try to characterize the result of the substitution process slightly more formally. Metatheorem 4.3.1 describes how the substitution postprocessing algorithm removes closures in the result. Notably, this states that a closure exists in the postprocessed result if and only if the closure's expression is a hole. This is consistent with what we expect in Hazelnut Live, where closures did not exist outside of hole closures. We can justify this by performing induction on the postprocessing rules; it is clear that all closures are eliminated using the

$\text{PPI}_{[]} \text{Closure}$ rule, and closures are only introduced using the $\text{PPO}_{[]} \text{EHole}$ and $\text{PPO}_{[]} \text{NEHole}$ rules.

Metatheorem 4.3.2 states that the postprocessed result of evaluation with environments is the same as the result of evaluation with substitution, as presented in Hazelnut Live. We provide an intuitive justification for this. First, we check that the evaluation rules for the environment model are correct, and this is easy due the similarity to the evaluation rules for evaluation with substitution and evaluation with environments. The difference in the result lies only outside the evaluation boundary: holes and variables may not be correctly bound outside the environment boundary. We then perform induction using the postprocessing rules to ensure that variables and hole environments are properly looked up outside the evaluation boundary. Metatheorem 4.2.1 states that the postprocessing algorithm will have the necessary environments to perform the substitution pass. Metatheorem 4.3.1 affirms that there will be no stray closures remaining in the result except hole closures, as non-hole closures did not exist in evaluation with substitution.

Metatheorem 4.3.1 (Substitution postprocessing closures). *If $\sigma \vdash d \Downarrow d_1$ and $d_1 \uparrow_{[]} d_2$, then:*

1. If $[\sigma]d_3 \in d_2$, then $d_3 = \langle \rangle^u$ or $d_3 = \langle d \rangle^u$.
2. If $d_3 = \langle \rangle^u$ or $d_3 = \langle d \rangle^u$, then $d_3 \subset [\sigma]d_3$.

Metatheorem 4.3.2 (Evaluation with environments correctness). *Let \Downarrow_s be the evaluation semantics described by Hazelnut Live [2]. Then if $\sigma \vdash d \Downarrow d_1$ and $d_1 \uparrow_{[]} d_2$, and if $d \Downarrow_s d_3$, then $d_2 = d_3$.*

4.3.3 Post-processing memoization

There is repeated postprocessing if the same closure environment is encountered multiple times in the evaluation result. If we can identify and look up environments, then we can memoize their postprocessing.

Modifications to the environment datatype

Memoization of environments requires a unique key for each environment. The existing environment type `Environment.t` is a map $\sigma : x \mapsto d$. We introduce a new environment type `EvalEnv.t`¹⁴ that is the product of an identifier and the variable map $\sigma : (\text{id}_\sigma, x \mapsto d)$, in which id_σ indicates a unique environment identifier.

To ensure that there is a bijection between environment identifiers and environments, a new unique identifier must be generated each time an environment is extended. An instance of `EvalEnvIdGen.t` is used to generate a new unique identifier, and is required as an additional argument to functions in the `EvalEnv` module that modify the environment¹⁵.

Note that while physical identity may be used to distinguish between different environments, it is difficult to use for efficient lookups due to the abstraction of pointers in a high-level language like OCaml or Javascript. We may think of numeric identifiers (in general) as high-level pointers. We may state this property of environment identifiers as Metatheorem 4.3.3, which allows us to use environment identifiers as a key for environments.

Metatheorem 4.3.3 (Use of id_σ as an identifier). *The mapping $i_\sigma : \sigma \mapsto \text{id}_\sigma$ that maps an environment (identified up to physical equality) to its assigned environment identifier is a bijection.*

We justify this by the construction of environment identifiers. $\sigma_i \neq \sigma_j$ implies that there is a series of modified environments $\{\sigma_i, \sigma_{i+1}, \dots, \sigma_{j-1}, \sigma_j\}$ (without loss of generality, assume σ_i is an earlier environment than σ_j). By construction, each element of the set $\{i_\sigma(\sigma_i), i_\sigma(\sigma_{i+1}), \dots, i_\sigma(\sigma_j)\}$ is unique. Thus $i_\sigma(\sigma_1) \neq i_\sigma(\sigma_2)$.

¹⁴This is the name in the current implementation (due to this environment type being specialized for evaluation), but perhaps a better name is `MemoEnv.t`.

¹⁵In the same manner as `MetaVarGen.t`, `EvalEnvId.t` is implemented as type `int` and `EvalEnvIdGen.t` is implemented as a simple counter. To keep the implementation pure, the instance of `EvalEnvIdGen.t` needs to be threaded through all calls of `Evaluator.evaluate` to avoid a global mutable state, and is discussed in section 8.2.

Modifications to the post-processing rules

During substitution postprocessing ($\uparrow\downarrow$), a mapping $\text{id}_\sigma \mapsto \sigma$ stores the set of substituted (postprocessed) environments. Upon encountering a closure in the evaluation result, it is looked up in this map. If it is found, the stored result is used. If it is not found, the environment is recursively substituted by applying $\uparrow\downarrow, i$ to each binding.

4.4 Implementation considerations

This section details various design decisions and tradeoffs of the current implementation; some parts of this may require an understanding of the hole closure numbering postprocessing step described in Chapter 5.

4.4.1 Data structures

As is common in functional programming, the most common data structures used are (linked) lists and maps (binary search trees). The standard library modules `List` and `Map` are used for these. In particular, the original implementation uses linked-lists for the implementation of environments, and we have not modified this decision. In Hazel, the hole closure storage data structures `HoleClosureInfo_.t` and `HoleClosureInfo.t` use a combination of maps and lists.

The only major change to the data structures is the switch from using linked lists (`VarMap.t`) as the backing store for environments to using a binary search tree representation (`VarBstMap.t`). This improves performance of operations on large environments.

Hashtables were not used at all in the implementation; their effect on performance is unknown and is reserved for future work. While they allow for amortized $O(1)$ operations, they are stateful and thus difficult to copy, and do not allow for the structural sharing memory optimization. Since immutable data structures are efficiently copied each time they are modified, the costs of introducing hashtables will likely outweigh the costs.

4.4.2 Additional constraints due to hole closure numbering

Section 5.4.1 introduces another postprocessing algorithm, which may be combined with substitution postprocessing. The introduction of hole closure parents in section 5.3.1 makes closure memoization more difficult for environments in non-hole closures. In particular, adding a new parent to a hole requires that the hole postprocessing (the hole closure numbering operation) be re-run on a hole. Memoizing the hole prevents a hole closure in an environment from being assigned multiple closure parents. To get around this, we propose a modified memoization routine in section 5.4.3 that only postprocesses environments when a hole or variable is reached, rather than when a closure is reached in postprocessing.

4.4.3 Storing evaluation results versus internal expressions

The evaluation takes as input an internal expression and returns the evaluated internal expression along with a final judgment (either `BoxedValue` or `Indet`).

The decision should be made whether to store this final judgment in the environment¹⁶. Storing the judgment allows us to simply use the stored value directly during evaluation, but requires much boxing and unboxing in other cases (e.g., during postprocessing). On the other hand, not storing the judgment is cleaner when used outside of evaluation, but requires recalculation of the final judgment during evaluation upon lookup¹⁷. The decision is somewhat arbitrary but may have small effects on the evaluation performance and elegance of implementation.

¹⁶In other words, we need to decide whether `EvalEnv.t` should be a mapping from variables to `EvalEnv.result` (including final judgment) or from variables to `DHExp.t`.

¹⁷Recalculating the final judgment means re-evaluating the expression upon variable lookup, since the `Evaluator.evaluate` function currently performs the evaluation and final judgments. This should not be an expensive operation since the value should already be final and cannot make any evaluation steps, but still may require several calls to evaluate.

Chapter 5

Identifying hole closures by physical environment

5.1 Rationale behind hole instances and unique hole closures

Consider the program displayed in Listing 2. The evaluation result of the program is

$$[a \leftarrow [\emptyset] \textcolor{violet}{\textcircled{1}}, x \leftarrow 3] \textcolor{violet}{\textcircled{2}} + [a \leftarrow [\emptyset] \textcolor{violet}{\textcircled{1}}, x \leftarrow 4] \textcolor{violet}{\textcircled{2}}$$

Note that the two instances of $\textcolor{violet}{\textcircled{2}}$ have different environments, and we thus distinguish between the two occurrences of $\textcolor{violet}{\textcircled{2}}$ as separate *instances* of a hole. However, note that while there are also two instances of the hole $\textcolor{violet}{\textcircled{1}}$ in the result, these share the same (physically equal) environment. No matter what expression we fill hole $\textcolor{violet}{\textcircled{1}}$ with (for example, using the fill-and-resume operation) the hole will evaluate to the same value. This differs from the hole $\textcolor{violet}{\textcircled{2}}$, whose filling may cause different instances to evaluate to different values due to non-capture-avoiding substitution. For example, filling hole $\textcolor{violet}{\textcircled{2}}$ with the expression $x + 2$ will cause the instances to resolve to 5 and 6, respectively.

```
let a = ()1 in
let f = λ x . { ()2 } in
f 3 + f 4
```

Listing 2: Illustration of hole instances

The current implementation assigns an identifier i to each instance of a hole, and the instance number is unique between all instances of a hole. While this makes perfect sense for $()^2$, the assignment of two separate holes to $()^1$ may confuse Hazel users, since these hole instances are identical and filling them with any value will result in the same value. The solution is to unify all instances of a hole which share the same (physically equal) environment, and thus identify hole instances by hole number and environment. A set of hole instances that share the same environment will be called a *unique hole closure*, simply *hole closure*¹, or *hole instantiation*.

To illustrate why physical equality is used to identify environments, consider the case shown in Listing 3. This simpler program evaluates to

$$[x \leftarrow 2] ()^1 + [x \leftarrow 2] ()^1$$

In this case, hole 1 has two instances with two environments with structurally equal bindings. If the argument to the second invocation of f is changed to 3, then the holes will have different environments and may thus fill to different values. This may be confusing to the Hazel user; what appears to be a single hole closure is actually two different hole closures which incidentally have the same values bound to its variables.

An intuitive way of understanding the use of physical equality is that separate *instantiations* of the same hole should be distinguished. This is highly related to function applications.

A hole may only appear multiple times in the result in two different ways: it may exist in

¹"Hole closure" also is used to describe the generalized closure around hole expressions as described in Chapter 4. Here we are referring to the set of instances of the same hole that share the same physical environment. Hence we call this interpretation "unique hole closure" to distinguish it from the former interpretation, but the interpretation should be clear from context.

```
let f = λ x . { 01 } in
f 2 + f 2
```

Listing 3: Illustration of physical equality for environment memoization

the body of a function that is invoked multiple times (multiple hole instantiations), or it may appear in a hole that is referenced from other holes (shared hole instantiation). The former leads to multiple hole closures, while the latter leads to a single hole closure.

5.2 The existing hole instance numbering algorithm

Hole numbering is a process that follows evaluation and operates on the evaluation result². It assigns a hole instance number to each hole. Hole instances are briefly motivated in Hazelnut Live, but the algorithm for hole numbering algorithm was not. We will provide a brief high-level description of it here, and then provide the inference rules for our own implementation. It is a breadth-first search of the result, recursing through holes. When a hole is encountered, it is assigned a unique hole instance number³ and added to a data structure `HoleInstanceStateInfo.t` that keeps track of all hole instances. Each hole instance's hole number, hole instance number, hole closure environment, and path⁴ is stored in this data structure. The `HoleInstanceStateInfo.t` is in turn stored in the `Result.t` that stores all of the information about an evaluated program. The primary use of `HoleInstanceStateInfo.t` is for the context inspector. With this data structure, users may easily iterate all instances of a selected hole, examine the hole path of a selected hole, examine the environment of a selected hole, or navigate to another hole instance.

²The function in the existing codebase that performs hole renumbering is `Program.renumber`. We may refer to it throughout this text as “hole numbering,” “hole renumbering,” or “hole tracking.”

³I.e., each hole instance is uniquely identified by the pair of identifiers (u, i) . The hole instance number only has to be unique out of the hole instances for a particular hole u .

⁴The path of a hole is the recursive list of hole parents that must be traversed in order to reach a hole. In other words, this is the path to a hole if we envision the result expression as a tree, in which each hole is a node that fathers all of its variable binding expressions.

5.3 Issues with the current implementation

Consider the program shown in Listing 4. A performance issue appears with the existing evaluator with this program⁵. As we increase the number of consecutive `let` expressions, we get an exponential slowdown that makes evaluation impractical for $n > 10$. The results of running this program for several values of n is shown in tabular form in Table 7.2 and graphically at Figure 7.4a.

For now, let us consider the case when $n = 3$. When evaluating with environments⁶, the result is shown in Figure 5.1.

The program slowdown happens in the hole numbering process. Recall from section 5.2 that the hole numbering process is a simple tree traversal algorithm. Thus, each time a hole (with the same environment) is encountered, it and all of its descendant holes will be given more hole instance numbers. This leads to the hole numbering shown in Figure 5.2. We see that there are four instances of hole 1, two instances of hole 2, and one instance of hole 3. In sum, we see that there are eight total hole instances. The number of holes increases by powers of two. As n increases, the total number of holes (including the instance of last hole) will be exactly 2^n .

Clearly this is undesirable from an efficiency perspective. It is also undesirable from the perspective that there is only one instantiation of each of the holes. While there are multiple paths to each node, we would like to change the representation to match that of the unique hole closures or hole instantiations as described in section 5.1.

5.3.1 Hole instance path versus hole closure parents

Visually, we would like to change the hole tracking to use a representation more similar to Figure 5.1 rather than that of Figure 5.2. In the old representation, each hole instance is

⁵This was first brought to attention by a GitHub issue at <https://github.com/hazelgrove/hazel/issues/536>.

⁶When evaluating using the substitution model, evaluation also slows down exponentially, because the variables are eagerly substituted into the hole environments. We do not have a performance issue with evaluation with environments because of lazy variable lookups.

```

let a = ()1 in
let b = ()2 in
let c = ()3 in
let d = ()4 in
let e = ()5 in
let f = ()6 in
let g = ()7 in
...
let x = ()n in
()n+1

```

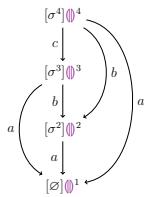
Listing 4: A Hazel program that generates an exponential (2^N) number of total hole instances

Figure 5.1: Structure of the result of the program in Listing 4

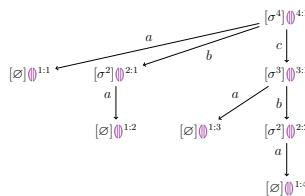


Figure 5.2: Numbered hole instances in the result of Listing 4

uniquely identified by a hole number and hole path.

In the new representation, hole instantiations are uniquely identified by hole number and environment, and are not uniquely identified by a path anymore. Thus, for each hole closure we instead keep track of a list of its parent holes.

We note that these two representations of a graph are equivalent⁷, assuming that nodes sharing an environment are considered to be physically equal. The first describes the path to each node, while the latter is an adjacency list representation. Either representation of a graph can be used to construct the other, but the latter is much more efficient in the case of a dense graph.

Changing the structure from using hole paths to hole parents forces a minor change to the Hazel UI. When a user selects a hole, rather than showing the path to the hole, the list of parents to the hole are shown instead.

5.4 Algorithmic concerns and a two-stage approach

To efficiently build the new hole-tracking data structure, we expect to have a fast lookup of hole numbers and environment identifiers. On the other hand, we want the interface of this data structure to be similar to the interface of `HoleInstanceStateInfo.t`: the user should be able to efficiently look up environments by hole number and hole closure number.

To efficiently handle both of these desired properties, we require two different data structures. The first is an auxiliary data structure `HoleClosureInfo_t` that is a map $H : (u, \sigma) \mapsto (i, p)$ where p denotes the the parents of a hole closure. The second is the data structure that will be used for the context inspector and hole closure lookups, `HoleClosureInfo.t`, that is a map $H : (u, i) \mapsto (\sigma, p)$. The maps are implemented as binary search trees for efficient lookups and updates⁸. The first stage of this algorithm is to build

⁷This structure is more specifically a join-semilattice.

⁸Note that this is one of the few places where a hashtable implementation is appropriate in the context of this project, since we do not copy these data structures. However, there will likely not be a major performance benefit; the main benefit lies in memoizing environments.

the `HoleClosureInfo_.t`; the second stage is to convert it to a `HoleClosureInfo.t`.

For convenience, we do not use two different symbols for these two data structures; the difference is purely an implementation detail regarding the construction of the data structure. The conversion from `HoleClosureInfo_.t` to `HoleClosureInfo.t` is trivial and will not be described here in detail. It simply involves iterating over the unique hole closures and changing the mapping to be indexed by hole number and hole closure number.

5.4.1 The hole numbering algorithm

The hole numbering algorithm is shown in Figure 5.3. Constants and variables are left unchanged by the hole numbering algorithm. For ordinary expressions with subexpressions, the algorithm recurses through subexpressions.

For hole expressions that have not been encountered before, the hole number and environment do not exist in H . A hole closure number $i = \text{hid}(H, u)$ is generated to be unique out of hole closures for the hole u . We recursively postprocess the environment. Then the hole closure is inserted into H , along with the postprocessed environment.

For hole expressions that have been encountered before, the hole number and environment do exist in H . We can use this to look up the hole closure number i , postprocessed environment σ' , and list of parents $\{p_i\}$ for this hole closure. We update the list of parents to include the current parent p , and return the hole numbered with i .

This algorithm memoizes environments by storing them in H . Thus when a encountered that has already been postprocessed is encountered, it uses the looked-up environment rather than re-postprocessing the environment. This memoization is not necessary to the algorithm.

5.4.2 Hole closure numbering order

The order of the numbers assigned to hole closures is not specified in the algorithm shown in Figure 5.3, but it is a consideration in the implementation. In the existing implementation, the evaluation result is traversed in a breadth-first search (BFS) order. On the other hand,

$\boxed{H, p \vdash d \uparrow_i; d' \dashv H'} d \text{ gets renumbered to } d'$
$\frac{}{H, p \vdash c \uparrow_i c \dashv H} \text{PP}_i\text{Const}$
$\frac{}{H, p \vdash x \uparrow_i x \dashv H} \text{PP}_i\text{Var}$
$\frac{H, p \vdash d \uparrow_i d' \dashv H'}{H, p \vdash \lambda x : \tau. d \uparrow_i \lambda x : \tau. d' \dashv H'} \text{PP}_i\text{Lam}$
$\frac{H, p \vdash d_1 \uparrow_i d'_1 \dashv H' \quad H', p \vdash d_2 \uparrow_i d'_2 \dashv H''}{H, p \vdash d_1 d_2 \uparrow_i d'_1 d'_2 \dashv H'} \text{PP}_i\text{Ap}$
$\frac{H, p \vdash d \uparrow_i d' \dashv H'}{H, p \vdash d : \tau \uparrow_i d' : \tau \dashv H'} \text{PP}_i\text{Asc}$
$\frac{(u, e) \notin H \quad i = \text{hid}(H, u) \quad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma')}{H, (u, i) \vdash \sigma \uparrow_i \sigma' \dashv H'' \quad H, p \vdash [\sigma^e] \textcolor{purple}{\emptyset}^u \uparrow_i [\sigma'^e] \textcolor{blue}{\emptyset}^{u,i} \dashv H''} \text{PP}_i\text{EHoleNew}$
$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma^e) \quad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e)}{H, p \vdash [\sigma^e] \textcolor{purple}{\emptyset}^u \uparrow_i [\sigma'^e] \textcolor{blue}{\emptyset}^{u,i} \dashv H''} \text{PP}_i\text{EHoleFound}$
$\frac{(u, e) \notin H \quad i = \text{hid}(H, u) \quad H, (u, e) \vdash \sigma \uparrow_i \sigma' \dashv H' \quad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma') \quad H'', p \vdash d \uparrow_i d' \dashv H'''}{H, p \vdash [\sigma^e] \textcolor{purple}{[d]}^u \uparrow_i [\sigma'^e] \textcolor{blue}{[d']}^{u,i} \dashv H'''} \text{PP}_i\text{NEHoleNew}$
$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma^e) \quad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e) \quad H'', p \vdash d \uparrow_i d' \dashv H'''}{H, p \vdash [\sigma^e] \textcolor{purple}{[d]}^u \uparrow_i [\sigma'^e] \textcolor{blue}{[d']}^{u,i} \dashv H'''} \text{PP}_i\text{NEHoleFound}$
$\boxed{H, p \vdash \sigma \uparrow_i; \sigma' \dashv H'} \sigma \text{ gets renumbered to } \sigma'$
$\frac{}{H, p \vdash \emptyset \uparrow_i \emptyset \dashv H} \text{PP}_i\text{EnvTriv}$
$\frac{H, p \vdash \sigma \uparrow_i \sigma' \dashv H' \quad H', p \vdash d \uparrow_i d' \dashv H''}{H, p \vdash \sigma, x \leftarrow d \uparrow_i; \sigma', x \leftarrow d' \dashv H''} \text{PP}_i\text{Env}$

Figure 5.3: Hole closure numbering postprocessing semantics

our implementation is implemented using a simpler depth-first search (DFS) order. This changes the order that hole closures are encountered and numbered. While the hole closure number is not specified explicitly ordered value, certain orderings may be more intuitive to the user. A choice of explicit hole numbering order is left to future work.

5.4.3 Unification with substitution postprocessing

The overall postprocessing operation is shown in Figure 5.4. The postprocessing is the composition of the substitution postprocessing and the hole numbering postprocessing.

We can combine the two postprocessing steps into a single pass. Since the renumbering step leaves most expressions unchanged, it is convenient to incorporate the hole numbering rules into the closure and hole rules for substitution postprocessing. The reference implementation uses this single-pass postprocessing.

There is a nuance here: the hole numbering step limits the use of memoization of environments in the postprocessing process. In particular, a complexity arises because we wish to update the list of parents of a hole each time a hole is encountered, forcing us to re-postprocess environments. Thus we require that closure environments are always “shallowly” postprocessed. This means that environments are postprocessed whenever a hole is encountered, rather than whenever a closure is encountered. Lastly, since the λ -closure postprocessing requires that the environment be postprocessed before performing a variable substitution, environments must also be postprocessed whenever a (bound) variable is encountered. In summary, the $\text{PPI}_{\parallel}\text{Closure}$ rule (outside the evaluation boundary) does not postprocess the closure’s environment, and the $\text{PPI}_{\parallel}\text{Var}$ and $\text{PPI}_{\parallel}(N)\text{EHole}$ rules (inside the evaluation boundary) postprocess the environment. There is also no need to match the closure around the hole, since closures and the environment will be handled by the substitution algorithm.

This “fix” is not ideal because of the added complexity and confusion, but it does mostly memoize the postprocessing process. We leave a more elegant implementation of the post-

$$\boxed{
 \begin{array}{c}
 \boxed{d \triangleleft (H, d')} \text{ } d \text{ postprocesses to } d' \text{ with hole closure info } H \\
 \frac{d \uparrow\!\!\uparrow d' \quad \emptyset, \emptyset \vdash d' \uparrow_i; d'' \dashv H}{d \uparrow\!\!\uparrow d'' \dashv H} \text{ PP-Result}
 \end{array}
 }$$

Figure 5.4: Overall postprocessing judgment

processing step for future work. A possible method is proposed in Section 9.1. However, we note that we do not note any slowdown with postprocessing when we perform our empirical performance evaluation in section 7.2.

5.4.4 Characterizing hole numbering

Metatheorem 5.4.1 summarizes the grouping of hole instances with the same environment (hole instantiations) into the same hole closure. This checks that the numbering follows the current implementation of hole instantiations.

Metatheorem 5.4.1 (Hole numbering postprocessing). *Let $\emptyset \vdash d \Downarrow d_1$ and $d_1 \uparrow_i; d_2 \dashv H$.*

1. If $[\sigma] \text{ } \textcolor{purple}{\textcircled{0}}^{ui} \in d_2$ and $[\sigma] \text{ } \textcolor{purple}{\textcircled{0}}^{ui'} \in d_2$, then $i = i'$.
2. If $[\sigma] \text{ } \textcolor{purple}{\textcircled{0}}^{ui} \in d_2$ and $[\sigma] \text{ } \textcolor{purple}{\textcircled{0}}^{ui'} \in d_2$, then $i = i'$ and $d_3 = d_4$.

If we inductively follow the rules of the hole numbering algorithm, we expect that holes with the same physical environment will have the same hole closure number i . The case of non-empty hole has an additional clause about the nested expression inside the non-empty hole: we expect it to be the same if the environment is the same, since these are the same instantiation of the hole.

We note that we do not establish a correctness theorem like Metatheorem 4.3.2, since we introduce a new interpretation of hole closure numbers. This changes the result from that of Hazelnut Live.

5.5 Fast structural equality checking

After the hole instance numbering is solved, there is an additional performance issue that is related to a recursive traversal of the evaluation result. After evaluation and hole numbering, there is an additional step (located in `Model.update_program`) that compares two evaluation results (`Result.t`) using a structural equality check⁹.

This step is also very slow if repeated environments are re-traversed, so we memoize it by environments. We manually implement a structural checking algorithm, `DHExp.fast_equals`. For any leaf node (node with no subexpressions), the value of the node is compared for equality. For branch nodes (nodes with subexpressions), the nodes are equal if subexpressions are equal and if the node's properties are equal. Importantly, the equality check for environments is simply to check if the environment identifiers are equal.

This step assumes that checking the equality of environment identifiers is equivalent to checking the physical equality of environments. This statement is true by Metatheorem 4.3.3 for two environments in the same program evaluation, but this may not be the case for comparing environments across separate evaluations. Thus we also need to structurally check environments. Luckily, this is easily memoized so we only compare environments once.

We do not feel that it is necessary to write out the judgments for this equality checking, which are very similar to a simple recursive structural equality check. The operation of the algorithm and its correctness should be fairly intuitive by the description.

⁹This step in `Model.update_program` is used to check if the program result changes. There may be more efficient heuristics to detect a change in program output, such as comparing the programs' external expressions or detecting the type of edit action. However, we are simply concerned here with maintaining the original intent of comparing structural equality. Moreover, this memoized structural equality check may be useful whenever a structural equality check is required on expressions with environments.

Chapter 6

Implementation of fill-and-resume

6.1 Motivation

Consider the program shown in Listing 5. In this program, the calculation of $x = \text{fib } 30$ is arbitrarily chosen to represent a computationally-expensive operation. The result of this program is

$$[f \leftarrow [\emptyset] \lambda x. \{ \dots \}, x \leftarrow 832040] \oplus^1$$

Now, if we want to “fill” hole 1 with the expression $x+2$, then it would seem wasteful to have to re-compute the value of x . After all, the computed value bound to x is exactly the same. Moreover, we realize that the evaluation result stores the computed value of x in the hole’s closure’s environment. Rather than re-evaluating the original program, we may instead *fill* the hole in the evaluated program result, and then *resume* evaluation.

$$\begin{aligned} & [f \leftarrow [\emptyset] \lambda x. \{ \dots \}, x \leftarrow 832040] (x + 2) \\ & \quad 832040 + 2 \\ & \quad 832042 \end{aligned}$$

```

let f : Int → Int =
λ x . {
  case x of
    | 0 => 0
    | 1 => 1
    | n => f (n - 1) + f (n - 2)
    end
}
in x = f 30
in ()1

```

Listing 5: A sample program with an expensive calculation stored in a hole's environment

Here we observe that the generalized closures surrounding holes and other stopped evaluations allow us to capture the environment for future computation.

This is the *fill-and-resume (FAR)* operation that is described in Hazelnut Live [2]. It is described in terms of a substitution-based evaluation semantics, and with respect to its theoretic foundations in contextual modal type theory (CMTT). We provide the first implementation of such an operation. Hazelnut Live also describes a practical problem with fill operations that take place over multiple edit actions (*n*-step FAR), with the suggestion to “cache more than one recent edit state to take full advantage of hole filling.” We present a structural diffing¹ algorithm in section 6.2.1 that easily allows us to detect and fill a hole from an arbitrary past edit state.

6.2 The FAR process

The fill-and-resume process can be broken into the following sequence.

1. Obtain a previous edit state with which to fill from the model.
2. Determine whether a fill operation is appropriate. If it is not, perform regular evaluation of the program, and do not continue to the following steps.

¹“Diffing” taken to mean the action of performing a (structural) diff operation between two edit states.

3. If the fill operation is valid, then obtain the fill parameters (the internal expression to fill, and the hole number from the previous edit state with which to fill).
4. **Fill.** Pre-process the evaluation result to prepare for (re-)evaluation.
5. **Resume.** Re-evaluate the filled expression.
6. Post-process the evaluation result for display purposes.
7. Update the model with the evaluation results.

These steps will be described in greater detail in the following sections.

6.2.1 Detecting the fill parameters via structural diff

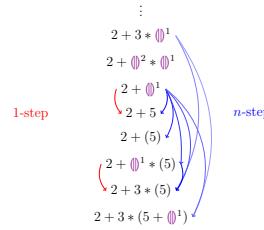
Following the notation from [2], the fill operation $\llbracket d_{fill}/u_{fill} \rrbracket d_{result}$ indicates the fill of hole u_{fill} with expression d_{fill} in the expression d_{result} . d_{result} is the past program result with which to fill. We need a method to determine the fill parameters u_{fill} and d_{fill} .

A naïve algorithm for detecting fill parameters

One way to approach the problem of obtaining the hole number and filled expression is at action time. When constructing an expression, we can check if the cursor lies in a hole (either directly in an empty hole, or if there is an ancestor non-empty hole). When deleting an expression, we can check if the cursor lies in a non-empty hole. However, this method is somewhat short-sighted. What happens if we wish to make multiple edits, e.g., fill a hole with the number 12? Then there are two actions, and the second action is not a hole fill action.

1

12

Figure 6.1: 1-step vs. n -step FAR

We may remedy this specific case by grouping together consecutive construction actions². However, we may also consider more complicated edit sequences that involve movement and deletion actions, potentially outside of the hole. Consider the edit sequence shown in Figure 6.1. The final edit state in this sequence is actually a valid fill of the first edit state of the sequence³, such that $u = 1$ and $d = 5 + (())^1$. However, an algorithm to trace the edit actions to determine that this is a valid fill may be difficult, since there is an incomprehensible mix of construct, delete, and movement edit actions. Even worse, the edits traverse outside the original hole, which likely makes the algorithm intractable.

This method is only tractable for 1-step FAR or simple cases of grouped edit actions. We wish for a more robust yet still simple solution that is independent of the edit sequence between two states.

Structural diffing between two edit states

Instead of observing the edit action, we may instead attempt to find the root of the difference between any two edit states, and determine if that is the the difference gives valid fill

²Grouping together of actions is already performed to some level by the undo history, for visual purposes.

³There are actually multiple valid fill operations here. Another valid fill operation occurs between the third edit state and all of the following edit states.

$d_1 \not\sim d_2$	Some (non-empty) diff between d_1 and d_2 .
$\frac{d_1 \triangleright d_2}{d_1 \not\sim d_2}$	SDiffNFDiffSome
$\frac{d_1 \not\sim d_2}{d_1 \triangleright d_2}$	SDiffFFDiffSome
$\frac{d_1 \not\sim d_2}{d_1 \triangleright d_2}$	SDiffSomeNFDiff
$\frac{d_1 \not\sim d_2}{d_1 \triangleright d_2}$	SDiffSomeFDiff
$d_1 \sim d_2$	Any (possibly-empty) diff between d_1 and d_2 .
$\frac{d_1 \triangleright_d d_2}{d_1 \triangleright_d d_2}$	ADiffNoDiffAny
$\frac{d_1 \triangleright_d d_2}{d_1 \not\sim d_2}$	ADiffNFDiffAny
$\frac{d_1 \triangleright_d d_2}{d_1 \triangleright_d d_2}$	ADiffFFDiffAny
$\frac{d_1 \triangleright_d d_2}{d_1 \triangleright_d d_2}$	ADiffAnyNoDiff
$\frac{d_1 \not\sim d_2}{d_1 \triangleright_d d_2}$	ADiffAnyNFDiff
$\frac{d_1 \not\sim d_2}{d_1 \triangleright_d d_2}$	ADiffAnyFDiff

Figure 6.2: Structural diffing abbreviated judgments

parameters. This has the benefit of being a relatively simple algorithm, while overcoming the limitation of the previous method and allowing for n -step FAR.

The structural diff algorithm takes two expressions as input and returns one of three diff judgments⁴. $d_1^- \triangleright d_2^-$ indicates *no diff* between d_1 and d_2 . $d_1^- \triangleright d_2^-$ indicates a *non-fill diff* from d_1 to d_2 . $d_1^- \triangleright_d^+ d_2^-$ indicates a *fill diff* of hole u with expression d from d_1 to d_2 .

We also define two shorthand operators for notational convenience. $d_1^- \not\sim_{d^+} d_2^-$ indicates *some (non-empty) diff* from d_1 to d_2 , which may or may not be a fill difference. $d_1^- \triangleright_{d^+} d_2^-$ indicates *any diff* (potentially no diff). Both notations are used to avoid writing multiple similar rules, where the only change in the rules is diff judgment type. The behavior of these notations is described in Figure 6.2.

For convenience, we define the judgment⁵ $d_1^- \sim d_2^-$ to mean that d_1 and d_2 are of the same *expression form*. We use the term *expression form* or *expression variant* to indicate

⁴The following notations for diffing are chosen somewhat arbitrarily. The triangle seems appropriate because it has a variant with an equals bar (\triangleright), as well as a “no fill” (\triangleright) and “fill” variant (\triangleright_d). The triangle is also horizontally asymmetric, which mirrors the fact that the diff relation is asymmetric.

⁵The relation \sim is already defined to mean type consistency when applied to types. This interpretation applies when the relation is applied to internal expressions.

$d_1 \sim d_2$	d_1 has the same expression form as d_2
$c_1 \sim c_2$	FEqConst
$x_1 \sim x_2$	FEqVar
$\lambda x. d_1 \sim \lambda x. d_2$	FEqLam
$e_1 e_2 \sim e'_1 e'_2$	FEqAp
$e : \tau \sim e' : \tau'$	FEqAsc
$\langle \rangle^u \sim \langle \rangle^{u'}$	FEqEHole
$\langle d \rangle^u \sim \langle d' \rangle^{u'}$	FEqNEHole

Figure 6.3: Form equality judgment

the variant types of `DHExp.t`. For example, empty holes and constants of the base type are different expression forms. Empty holes and non-empty holes are also different forms per the grammar. The rules shown in Figure 6.3 should require no further explanation.

We divide up the structural diffing algorithm into cases based on expression forms.

Expressions with different forms If the two expressions have different forms, then the current node is necessarily the diff root. It is a fill diff iff the left expression is a hole (DFEqEHole, DFNEqNEHole, DFNEqNonHole).

These rules are shown in Figure 6.4.

Expressions with the same non-hole form If the two expressions have the same form, then we need to check the root node and its subexpression(s). For expressions with no subexpressions, there is a non fill diff iff the expressions differ (DFEqConstNEq, DFEqConstEq, DFEqVarNEq).

For expressions with a single subexpression, we first check if there are any differences, ignoring the subexpression. If there is a difference, then the current node is the non fill diff root. Otherwise, we pass through the diff from the child node (DFEqLamNEq₁, DFEqLamNEq₂, DFEqLamEq, DFEqAscNEq, DFEqAscEq).

The last case to check for non-hole expressions are expressions with more than one subexpression. In this case, we first check if there exist any differences outside the subexpressions, which would result in a non fill diff rooted at the current node. Otherwise, if there are no subexpression diffs, then the result is no diff. If more than one subexpression has a diff, then the diff is a non fill diff rooted at the current node. The last case is when exactly one child has a diff, which would be passed through. This is the case for the binary function application expression form (DFEqApEq_1 , DFEqApEq_2 , DFEqApEq_3 , DFEqApEq_4).

In the minimal λ -calculus grammars specified for Hazel, the only expression form of plural subexpression arity is function application, but the following description extends to higher numbers of subexpressions (such as `case` expressions with arbitrary numbers of rules).

These rules are shown in Figure 6.5.

Expressions with the same hole form The last case to consider is the comparison of two hole expressions of the same form. The empty hole case is very similar to the nullary subexpression case (DFEqEHoleNEq , DFEqEHoleEq_1). The non-empty hole case is very similar to the unary subexpression case (DFEqNEHoleNEq , DFEqNEHoleEq_1 , DFEqNEHoleEq_2), except for a special rule that propagates non-fill diffs upwards to be a fill diff rooted in the current hole (DFEqNEHoleEqProp). This allows for diffs that are not rooted directly in a hole to be filled in their nearest non-empty hole parent node.

These rules are shown in Figure 6.6.

The diff algorithm begins by performing the structural diff between the elaborated program state of a past edit state d_{old} and the current edit state d_{cur} . A FAR operation is only valid if the diff judgment is a fill diff $d_{old} \triangleright_d^g d_{cur}$, which gives us the FAR parameters u and d .

$\frac{\emptyset^u \not\sim d_2}{\emptyset^u \blacktriangleright_{d_2}^u d_2}$ DFNEqEHole	$\frac{(\langle d \rangle)^u \not\sim d_2}{(\langle d \rangle)^u \blacktriangleright_{d_2}^u d_2}$ DFNEqNEHole
$\frac{d_1 \neq \emptyset^u \quad d_1 \neq (\langle d \rangle)^u \quad d_1 \not\sim d_2}{d_1 \triangleright d_2}$ DFNEqNonHole	

Figure 6.4: Structural diffling of different expression forms

$\frac{c_1 \neq c_2}{c_1 \triangleright c_2}$ DFEqConstNEq	$\frac{}{c \sqsupseteq c}$ DFEqConstEq	$\frac{x_1 \neq x_2}{x_1 \triangleright x_2}$ DFEqVarNEq
$\frac{}{x \sqsupseteq x}$ DFEqVarEq	$\frac{x_1 \neq x_2}{\lambda x_1 : \tau_1, d_1 \triangleright \lambda x_2 : \tau_2, d_2}$ DFEqLamNEq ₁	
$\frac{\tau_1 \neq \tau_2}{\lambda x : \tau_1, d_1 \triangleright \lambda x_2 : \tau_2, d_2}$ DFEqLamNEq ₂	$\frac{d_1 \triangleright_d^u d_2}{\lambda x : \tau, d_1 \triangleright_d^u \lambda x : \tau, d_2}$ DFEqLamEq	
$\frac{\tau_1 \neq \tau_2}{d_1 : \tau_1 \triangleright d_2 : \tau_2}$ DFEqAscNEq	$\frac{d_1 \triangleright_d^u d_2}{d_1 : \tau \triangleright_d^u d_2 : \tau}$ DFEqAscEq	
$\frac{d_1 \sqsupseteq d'_1 \quad d_2 \sqsupseteq d'_2}{d_1 \ d_2 \sqsupseteq d'_1 \ d'_2}$ DFEqApEq ₁	$\frac{d_1 \not\sqsupseteq_d^u d'_1 \quad d_2 \not\sqsupseteq_d^u d'_2}{d_1 \ d_2 \sqsupseteq d'_1 \ d'_2}$ DFEqApEq ₂	
$\frac{d_1 \sqsupseteq d'_1 \quad d_2 \not\sqsupseteq_d^u d'_2}{d_1 \ d_2 \not\sqsupseteq_d^u d'_1 \ d'_2}$ DFEqApEq ₃	$\frac{d_1 \not\sqsupseteq_d^u d'_1 \quad d_2 \sqsupseteq d'_2}{d_1 \ d_2 \not\sqsupseteq_d^u d'_1 \ d'_2}$ DFEqApEq ₄	
$\frac{u \neq u'}{\emptyset^u \blacktriangleright_{\emptyset^{u'}}^u \emptyset^{u'}}$ DFEqEholeNEq	$\frac{}{\emptyset^u \sqsupseteq \emptyset^{u'}}$ DFEqEholeEq	
$\frac{u \neq u'}{(\langle d \rangle)^u \blacktriangleright_{(\langle d \rangle)^{u'}}^u (\langle d' \rangle)^{u'}}$ DFEqNEHoleNEq	$\frac{d \sqsupseteq d'}{(\langle d \rangle)^u \sqsupseteq (\langle d' \rangle)^u}$ DFEqNEHoleEq ₁	
$\frac{d \blacktriangleright_{d'}^{u'} d'}{(\langle d \rangle)^u \blacktriangleright_{d'}^{u'} (\langle d' \rangle)^u}$ DFEqNEHoleEq ₂	$\frac{d \triangleright d'}{(\langle d \rangle)^u \blacktriangleright_{(\langle d \rangle)^u}^u (\langle d' \rangle)^u}$ DFEqNEHoleEqProp	

Figure 6.5: Structural diffling of non-hole expressions

$$\begin{array}{c}
 \frac{u \neq u'}{\langle \emptyset \rangle^u \blacktriangleright_{\langle d \rangle^{u'}}^u \langle \emptyset \rangle^{u'}} \text{DFEqEHoleNEq} \quad \frac{}{\langle \emptyset \rangle^u \succeq \langle \emptyset \rangle^u} \text{DFEqEHoleEq} \\
 \frac{u \neq u'}{\langle d \rangle^u \blacktriangleright_{\langle d \rangle^u}^u \langle d' \rangle^{u'}} \text{DFEqNEHoleNEq} \quad \frac{d \sqsupseteq d'}{\langle d \rangle^u \geq \langle d' \rangle^u} \text{DFEqNEHoleEq} \\
 \frac{d \blacktriangleright_{\langle d \rangle^u}^{u'} d'}{\langle d \rangle^u \blacktriangleright_{\langle d \rangle^u}^{u'} \langle d' \rangle^u} \text{DFEqNEHoleEq2} \quad \frac{d \triangleright d'}{\langle d \rangle^u \blacktriangleright_{\langle d \rangle^u}^u \langle d' \rangle^u} \text{DFEqNEHoleEqProp}
 \end{array}$$

Figure 6.6: Structural diffing of hole expressions

Performance tradeoffs of the two detection algorithms

The efficiency of the naïve approach is $O(\log E)$, where E is the number of expression nodes in the program, if we assume that the depth of an expression node is logarithmic with respect to the total number of expression nodes. That algorithm only has to traverse up the ancestors to decide whether the edit lies in a hole.

The structural diff algorithm presented in this section is $O(E)$, since it traverses each node (once) until it finds a difference. However, if one travels backwards multiple edit states, then the cost is $O(SE)$, where S is the number of edit states compared using this algorithm. While this is much more expensive than the previous algorithm, we assume that the program size is relatively small, causing delay only on the order of milliseconds. However, it may be able to find a valid fill-and-resume in many more cases than the previous algorithm, potentially saving a much longer repeated evaluation time. An analysis of the tradeoff between the number of edit states S to search and the expected performance gain is left for future work.

6.2.2 “Fill”: pre-processing the evaluation result for re-evaluation

Before beginning the re-evaluation, we would like to substitute all instances of the hole in the previous evaluation result d_{result} with the substituted expression. Each time a the hole u_{fill} is encountered, it is replaced with the fill expression d_{fill} . This way, we can simply reinvoke

the evaluation function on the past program result and expect it to resume the evaluation.

There are a few nuances here that should be addressed. First of all, we acknowledge another benefit of the generalized closure variant. If the environment was still baked into the hole, and the hole was replaced with an expression, we would need an additional mechanism to remember the hole's environment. This becomes more messy when the hole doesn't lie directly in a hole closure (i.e., if the hole lies outside the evaluation boundary). Closures are simply recursed through in this pre-processing step, and their environments will still be available even when the hole gets replaced with the fill expression.

We note that the pre-processing acts on the un-post-processed previous evaluation result d_{result} . This is because the internal expression directly from evaluation and the result after post-processing have different properties or invariants. We do not want to invalidate the properties that are expected to be upheld during evaluation, such as the fact that the body of any λ -abstraction lies outside the evaluation boundary (whereas post-processing will modify function bodies).

Another issue to tackle is the problem that closures were previously considered to be final values, and would not be re-evaluated. Technically, we may re-evaluate closures; since the evaluation function is idempotent, this will not yield the incorrect result, but it is needlessly inefficient. However, we will need to re-evaluate closures during FAR re-evaluation, since the fill expression will necessarily lie within some closure.

For efficiency reasons, we will only want to re-evaluate all closures in the result exactly once. To do this, we set a flag for the closure that indicates that it should be re-evaluated. This preprocessing step will recurse through d_{result} and set the flag to true for all closures, which is characterized by Metatheorem 6.4.3. All closures that result from an evaluation judgment will have the flag set to false. Only closures with the re-eval flag set will be re-evaluated. Closures with the re-eval flag set to false will act as values and evaluate to themselves, which is the same as the original evaluation behavior described in section 4.2. We denote closures with the re-eval flag set to false using the established notation for closures

$[\sigma]d^6$, and denote closures with the re-eval flag set to true by $\llbracket \sigma \rrbracket d^7$.

Finally, we may consider the issue of filling multiple instances of the same hole closure⁸. If we substitute the hole, then we lose the information about the holes instance, and thus cannot memoize the evaluations of the same hole instance by environment number, which may cause the evaluation result to differ from an ordinary non-fill evaluation. A solution to this is to introduce another `DHExp.t` variant `FillExp(HoleClosureId.t, DHExp.t)` that indicates the hole closure number as well as the expression to fill. This will be denoted using a hole with a subscript $\langle d \rangle_i$. The preprocessing expression will fill a hole u_{fill} with this expression rather than the expression d_{fill} directly. During evaluation, this data structure will facilitate the memoization of hole closures.

6.2.3 “Resume”: Modifications to allow for re-evaluation

Re-evaluation during fill-and-resume is mostly the same as regular evaluation, but now we need to keep in mind the considerations from pre-processing the previous program result.

The updated evaluation rules for closures are shown in Figure 6.7. All closures in the evaluation result will have been marked for re-evaluation by the pre-processing step. This means that the closure environment will first be recursively re-evaluated, following by the closure body. This ensures that the entire program result is fully evaluated⁹. We introduce a (re-)evaluation judgment for environments $\sigma^- \Downarrow \sigma^+$, which simply maps the evaluate operation over the bindings of an environment. Closures with the re-eval flag set to false will have the regular evaluation rule.

Due to the recursive nature of re-evaluation of closure environments, we expect eval-

⁶This allows previous discussions of closures to remain valid, since they take the interpretation that the re-eval flag is set to false.

⁷Using the double-square bracket notation also reinforces the fact that re-evaluation is tied with fill-and-resume, which also uses a double-square bracket notation.

⁸Hole closure refers to the interpretation from section 5.1: instances of hole u_{fill} that share the same (physical) environment.

⁹Note that there is now an “inversion” of evaluation order, in that we cannot expect the environment to be fully evaluated before it is encountered in a closure. In an ordinary evaluation, we would expect all the bindings in the environment to have been evaluated before they have been stored in the environment.

$$\begin{array}{c}
 \frac{}{\sigma' \vdash [\sigma]d \Downarrow [\sigma]d} \text{EEClosure} \quad \frac{\sigma \Downarrow \sigma'' \quad \sigma'' \vdash d \Downarrow d'}{\sigma' \vdash [\sigma]d \Downarrow d'} \text{EREClosure} \\
 \frac{\sigma \Downarrow \sigma' \quad \emptyset \vdash d \Downarrow d'}{\sigma, x \leftarrow d \Downarrow \sigma', x \leftarrow d'} \text{EREEnv} \quad \frac{}{\emptyset \Downarrow \emptyset} \text{EREEnvNull}
 \end{array}$$

Figure 6.7: Revised evaluation rules for closures

$\boxed{\sigma, \rho \vdash d \Downarrow d' \dashv \rho}$ d evaluates to d' given environment σ and previous fills ρ

$$\frac{}{\sigma, (\rho, i \leftarrow d) \vdash \langle d' \rangle_i \Downarrow d \dashv \rho} \text{EFillMemoFound}$$

$$\frac{(i \leftarrow d) \notin \rho \quad \sigma, \rho \vdash d' \Downarrow d'' \dashv \rho'}{\sigma, \rho \vdash \langle d' \rangle_i \Downarrow d'' \dashv \rho', i \leftarrow d''} \text{EFillMemoNew}$$

$$\frac{d \neq \langle d \rangle_i \quad \sigma \vdash d \Downarrow d'}{\sigma, \rho \vdash d \Downarrow d' \dashv \rho} \text{EFillOther}$$

Figure 6.8: Fill-memoized re-evaluation

ation to reach each closure marked for re-evaluation exactly once. This is characterized in Metatheorem 6.4.4.

The other difference that we have to deal with are memoizing the evaluation of the filled hole expressions. Per the discussion of the pre-processing step, all filled expressions will exist in a wrapper that indicates the hole number. We may simply memoize the results by that hole number. This requires us to thread some state throughout our evaluation¹⁰.

In these judgments, we introduce a new *fill memoization context* $\rho : i \mapsto d$, a mapping of hole closure numbers to expressions¹¹. A new *fill-memoized evaluation judgment* $\sigma^-, \rho^- \vdash d^- \Downarrow d^+ \dashv \rho^+$ describes the evaluation with memoization of hole fills. When a hole closure number is encountered for the first time, the expression is evaluated normally and added to ρ ;

¹⁰We have the `EvalState.t` object for threading state through evaluation, which helps with this implementation.

¹¹The symbol ρ was chosen arbitrarily. It is simply the Greek letter before σ .

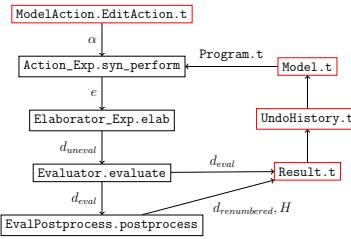


Figure 6.9: Previous action call graph

otherwise, the evaluated result of the fill is simply looked up from ρ . Note that memoization requires ρ to be threaded throughout the evaluation, i.e., it is treated both as an input and output of the evaluation judgment.

This fill-memoized evaluation judgment subsumes the normal evaluation judgment. For all non-hole-fill expressions, it performs the normal evaluation judgment and returns the fill memoization context unchanged.

6.2.4 Post-processing resumed evaluation

The postprocessing algorithm remains unchanged from before. Note that an evaluated program result should never include either of the new forms $(\langle d \rangle)_i$ or $[\sigma]d$; these should all have been encountered and evaluated out. In other words, the evaluation result from re-evaluation during a FAR should be indistinguishable from the evaluation result from a regular evaluation, and thus the postprocessing process is unchanged.

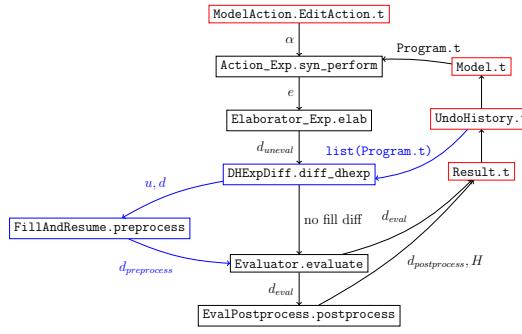


Figure 6.10: Current action call graph

6.3 Entrypoint to the FAR algorithm

An abstracted call graph for the action model (the process of responding to an action, up to evaluation) in Hazel is diagrammed in Figure 6.9. Red boxes indicate important data structures. Black boxes indicate important steps in the process of responding to an action. The lines between boxes roughly indicate the flow of the program and other important data structures related to the process.

An edit action triggers the bidirectional action semantics, which generates an updated program edit state. This is then elaborated to an internal expression, evaluated, and then postprocessed. The results of evaluation are stored in **Result.t**.

The updated abstract call graph is shown in Figure 6.10, with updated blocks shown in blue. Fill and resume is fundamentally an operation on internal expressions, so we attempt to detect a hole fill operation after elaboration of the current edit state. The structural diff operation requires information about at least one previous edit state from the model's

`UndoHistory.t`. The result of the previous evaluation is preprocessed, before being evaluated (using the updated evaluation judgments) and postprocessed as usual.

Since this is a very high-level view of the program, many details are abstracted out. For example, the evaluation function is memoized by the function `Program.get_result`, which facilitates the normal evaluation process. We will need to modify this memoization to also memoize the FAR evaluation results.

6.4 Characterizing FAR

We provide a characterization of the FAR process here via a set of metatheorems and informal justifications, as in previous sections on evaluation and the postprocessing process.

Hazelnut Live provides two metatheorems to describe the overall FAR process. We will restate them here for completeness. Proofs may be found in [2]. Metatheorem 6.4.1 describes the static semantics of filling. If the filled expression matches the assigned type of the hole in the hole context, then the type of the overall evaluation result should not change. This is a (type) preservation property. We assume that the premise is true by the action semantics, elaboration process, and structural diffling algorithm: any hole fill detected by structural diffling should be well-typed according to automatic empty-holes and casts inserted by the action semantics and elaboration process.

Metatheorem 6.4.1 (Filling (FAR static correctness)). *If $\Delta, u :: \tau[\Gamma'] ; \Gamma \vdash d : \tau$ and $\Delta ; \Gamma' \vdash d' : \tau'$ then $\Delta ; \Gamma \vdash [d'/u]d : \tau$.*

The dynamic correctness of fill-and-resume depends on the commutativity property stated in Metatheorem 6.4.2. Here, the notation $[d/u]d'$ is intended to mean the entire FAR process, not just the fill operation as we interpret it in the rest of this thesis. The proof is also provided in [2], as is the definition of the closure of evaluation stepping $d_1 \mapsto^* d_2$. An important point to note is that the proof of commutativity requires a functionally pure language (i.e., with no side effects) such as Hazel.

Metatheorem 6.4.2 (Commutativity (FAR dynamic correctness)). *If $\Delta, u :: \tau'[\Gamma'] ; \emptyset \vdash d_1 : \tau$ and $\Delta ; \Gamma' \vdash d' : \tau'$ and $d_1 \mapsto^* d_2$ then $[d'/u]d_1 \mapsto^* [d'/u]d_2$.*

In addition to the metatheorems governing the overall static and dynamic correctness of FAR, we may also characterize the intermediate steps, since we break up FAR into sequential fill and resume steps. Similar to Metatheorem 4.3.1, we characterize both the fill and resume steps by their effects on closures. This time, the focus is on re-evaluatable closures. We expect that the fill step marks all closures in the result for re-evaluation Metatheorem 6.4.3, and that the resume step reaches evaluation for all re-evaluation Metatheorem 6.4.4.

These metatheorems may not be necessary to prove commutativity, but they illustrate the behavior of the fill-and-resume steps: they demonstrate that re-evaluation reaches every expression within the evaluation boundary, including and especially any filled expressions. In other words, they check the proper coverage of the fill and resume steps when handling closures. These theorems are also both straightforwardly justified by the rules for the fill operation and evaluation.

Metatheorem 6.4.3 (Fill operation). *If $\emptyset \vdash d \Downarrow d_1$, and $[d_2/u]d_1 = d_3$, then $\bar{\beta}[\sigma]d_1 \in d_3$.*

Metatheorem 6.4.4 (Resume operation). *If $\emptyset \vdash d \Downarrow d_1$ and $[d_2/u]d_1 = d_3$ and $\emptyset \vdash d_3 \Downarrow d_4$ then $\bar{\beta}[\sigma]d_5 \in d_4$.*

6.5 FAR examples

6.5.1 Motivating example

We revisit the program from Listing 5, reproduced in Figure 6.11a. This hole fill operation is very simple, replacing a hole with a value. We also observe the behavior of the preprocessing operation, which marks closures for re-evaluation and encapsulates filled expressions with the hole closure number for memoization.

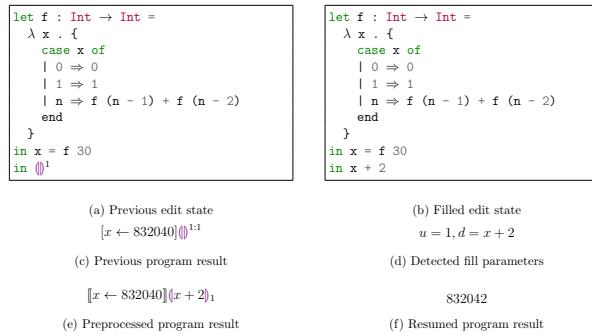


Figure 6.11: FAR simple example

6.5.2 Introducing and removing static type errors

Now, consider the program shown in Figure 6.12. In this fill operation, we introduce a new static type error (non-empty hole). The non-empty hole is inserted automatically during the action semantics and appears naturally in the diff, so we do not need to perform any special handling for it. Similarly, in the case of Figure 6.13, a non-empty hole is removed by filling with a type-consistent expression. This allows static type errors to be “fixed” and evaluation to resume past where it had stopped.

6.5.3 Example requiring recursive evaluation of closure environment

In Figure 6.14 we fill a hole that does not exist directly in the result, but exists in a hole closure environment. This illustrates the need to recursively re-evaluate closure environments.

(a) Previous edit state	(b) Filled edit state
$2 + [\emptyset] \textcolor{purple}{()^1}$	$u = 1, d = (\lambda x.x)^{1:1}$
(c) Previous program result	(d) Detected fill parameters
$2 + [\emptyset] (\lambda x.x)^{1:1}$	$2 + [\emptyset] [\emptyset] \lambda x.x^{1:1}$
(e) Preprocessed program result	(f) Resumed program result

Figure 6.12: FAR introduce static type error example

(a) Previous edit state	(b) Filled edit state
$2 + [\emptyset] ([\emptyset] \lambda x.x)^{1:1}$	$u = 1, d = 3$
(c) Previous program result	(d) Fill parameters
$2 + [\emptyset] (3)^1$	5
(e) Preprocessed program result	(f) Resumed program result

Figure 6.13: FAR remove static type error example

(a) Previous edit state	(b) Filled edit state
$[x \leftarrow [\emptyset] \textcolor{purple}{()^1}] \textcolor{purple}{()^2}$	$[x \leftarrow 2] \textcolor{purple}{()^2}$
(c) Previous program result	(d) Fill parameters
$[x \leftarrow [\emptyset] (2)] \textcolor{purple}{()^2}$	$[x \leftarrow 2] \textcolor{purple}{()^2}$
(e) Preprocessed program result	(f) Resumed program result

Figure 6.14: FAR fill hole in hole environment example

6.5.4 Hole fill expression memoization example

The program in fig. 6.15 shows the necessity of memoizing by hole closure number, in order to preserve the same result as if evaluating normally. In this example, we have two instances of hole 1 in the result. Since there is only one instantiation of hole 1, these are two instances of the same hole closure. If there were no memoization of filled expressions, then the filled expression $f 1$ would be evaluated twice. This is problematic because function application generates new environments (and new environment identifiers), so the two instances would have different closures and thus be considered two separate hole closures, even when they come from the same instantiation. Since the filled expression is memoized, the two instances of the hole have closures with the same environment identifier (indicated by the same superscript i on the closure environments).

The necessity of memoization in this case is related to the idea of the “inversion” of evaluation mentioned in section 6.2.3: during normal evaluation, we expect that a variable may be referenced in multiple places after it is evaluated. However, during a resumed evaluation, we may encounter multiple unevaluated instances of the same instantiation; the first time it is encountered and evaluated is the *de facto* “first instantiation” of that expression.

6.5.5 Noteworthy non-examples

Dynamic type errors

One may wonder if dynamic type errors (cast failure) have any nice relation to fill-and-resume, but it turns out that they are not treated much different than other non-hole expressions. We may not fill a cast failure directly, because it is not in a hole, and no environment is recorded for it¹². We may only remove a cast failure if it lies in a non-empty hole that is filled.

¹²With generalized closures, it is not hard to save the environment of a failed cast, but this is not very useful.

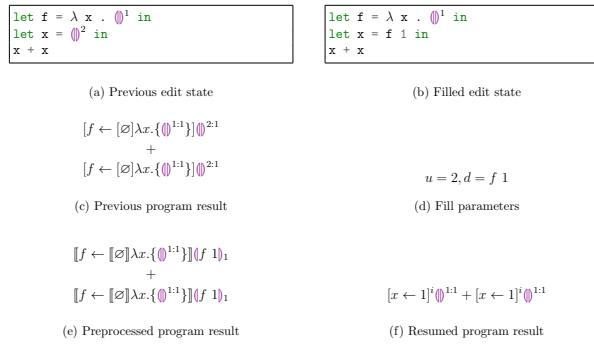


Figure 6.15: FAR hole closure memoization example

We may introduce new cast failures by filling an expression that is assigned to type hole (something of dynamic type). However, this is no different than introducing any well-typed expression and performing normal evaluation.

Thus the only interesting cases of introducing or removing holes lies in the case of static type errors (non-empty holes) as described previously.

Infix operators

Filling holes in an infix operator sequence may not result in a hole fill due to infix operator precedences, despite the initial appearance. Consider the example shown in Figure 6.16. Say we construct a binary plus operator in the hole. One might expect the fill operation to be $\llbracket \textcolor{purple}{\emptyset}^2 + \textcolor{purple}{\emptyset}^1 / 1 \rrbracket d_{result}$. However, we need to be careful: the multiplication operator has a higher precedence than the addition operator, causing the AST to have a different structure outside of where hole 1 was in the original edit state. It may be more clear if we write the latter

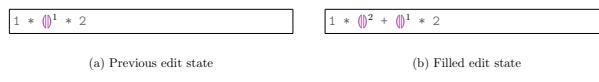


Figure 6.16: FAR infix operator fill

edit state with its implicit parentheses: $(1 * (0^2) + ((0^1 * 2)))$.

If, however, hole 1 in the original edit state was a parenthesized expression, and the sum expression shape was constructed inside the parenthesized expression, then the AST wouldn't change (as a parenthesized expression has higher precedence than the outer multiplication operation) and it would be a valid fill operation.

6.6 Tracking evaluation state

It may be useful to thread some state throughout the evaluation process. This means that changes to this state by an earlier invocation to evaluate will affect later evaluations.

The concept of keeping state has already appeared in several contexts. For postprocessing, the memoized hole closure info keeps track of seen environments (section 4.3.3). For evaluation, we keep track of the next unique environment identifier and memoize the evaluation of filled expressions (section 6.2.3). A theoretical discussion of statefulness is revisited in section 8.2.

This becomes relevant again (with respect to evaluation) because resuming evaluation requires us to store the state after evaluation, and restore the state after evaluation. Adding more state variables also makes the implementation messier by increasing the number of variables to pass to and return from each of the numerous calls to `Evaluator.evaluate`.

The nice solution to this is to group together all evaluation state under a single data structure, `EvalState.t`. This one state variable is passed around to all calls to evaluate, and is stored to and restored from `Result.t`, which stores all information about a program's evaluated result (including the evaluated expression and the hole closure information). Adding

state then becomes trivial, because we only have to modify this data structure rather than all of the calls to `Evaluator.evaluate`.

For sake of brevity, we will not update the judgments to include this state.

6.6.1 Step counting

With the `EvalState.t` data structure, it is now trivial to keep track of evaluation statistics. Sample statistics might include step counting and number of times a particular evaluation trace has been paused and resumed.

Step counting may be implemented simply by incrementing the count on each call to `Evaluator.evaluate`. This may be useful to stop long-running executions after a particular number of (possibly user-specified) execution steps in order to prevent program crashes. It is also useful for our purposes to track how efficient fill-and-resume is: during a fill operation, one may observe the difference in the number of steps before and after the resumed evaluation, and compare it to the case if the evaluation had begun from scratch.

6.7 Differences from the substitution model

The FAR operation is much simpler when evaluating with substitution rather than with environments. With substitution, we eliminate the need for pre- and post-processing steps, and dealing with re-evaluating closures. However, this comes at the cost of not being able to memoize repeated instances of the same hole closure. The original description of FAR from [2] is reproduced in Appendix B.2. The hole closure notation from [2], in which the hole closure is represented using a subscript, is also used here, since the substitution model doesn't have the concept of separate (generalized) closures.

What remains the same between the environment model and the substitution model is the need for an algorithm to detect the fill parameters, for which the structural diff algorithm presented in section 6.2.1 is still applicable.

The notation $\llbracket d/u \rrbracket d' = d''$ is used to indicate the FAR operation in the Hazelnut Live description, whereas we use it here to mean the fill operation only. This brings up the question of whether we can also perform FAR in a single step, similar to Appendix B.2, rather than breaking it into disjoint fill and resume passes. This is possible by either substituting the filled hole during evaluation. While this offers the same benefit of lazy substitution like evaluation with environments, there is some difficulty with lazily substituting a hole u if the fill expression contains a hole with the same hole number u .

Chapter 7

Evaluation of performance

To evaluate performance, benchmarks were carried out using the `TimeUtil.measure_time` utility. Benchmarks were carried out on Google Chrome 99 on Debian 11 on an Intel i3-2100 CPU. The times shown are a mean of three trials. Evaluation step counts are tracked in `EvalState.t` and count the number of calls to `Evaluator.evaluate` as described in section 6.6.1.

There are a number of factors that may affect the consistency of the elapsed time benchmarks. Such factors include the quality of JSOO-generated Javascript, specifics of the Chrome V8 Javascript engine, and millisecond precision of the timing function.

Evaluation of performance of the environment model of evaluation and of the memoized hole tracking are performed on the `dev` branch and the `eval-environment` branches of the Hazel repository. The latter branch implements our changes.

The demonstration of FAR is performed on the `fill-and-resume-backend` branch. This branch is based on the `eval-environment` branch, and implements a one-step FAR operation. This branch does not achieve parity with the theoretical model of FAR presented in this work; unfinished work and future improvements to the current implementation of FAR are described in section 9.2.

7.1 Evaluation of performance using the environment model

To evaluate the performance of evaluation using the environment model, we benchmark the performance of a computationally-expensive function, the tree-recursive Fibonacci function. This function is chosen because it is computationally expensive and does not have a deep recursion depth¹. It is also a complete program, i.e., it does not have holes and the hole renumbering and postprocessing steps are not of concern here.

For this experiment, the builtin variables and functions are removed. Restoring the builtins would be very similar to the second program variation described in section 7.1.2.

7.1.1 A computationally expensive fibonacci program

The quantitative results of this experiment are shown in Table 7.1. The results of evaluating Listing 6 for various values of n on the `dev` and `eval-environment` (abbreviated `e-e`) branches are shown in Figure 7.1. The `eval-environment` decreases evaluation time by a small, roughly-constant factor for all values of n .

7.1.2 Variations on the fibonacci program

We try out a few variations of the `fib(n)` function, shown in Listing 6. Results are collected for $n \in \{22, 23, 24, 25, 26\}$ ². The first variation is shown in Listing 7, which involves more global variables. The second variation is shown in Listing 8, in which an additional branch is added. This branch is never taken (as the third rule's pattern will always match). We vary the number of extra global variables and the length of the unevaluated branch.

However, the two variations show the difference between evaluation with environments

¹This is because Hazel does not implement tail-call optimization (TCO), and thus would overflow the stack for most iterative algorithms.

²These numbers were chosen somewhat arbitrarily. They are large enough to allow for reproducible results, and small enough to prevent excessively long runtimes.

```

let f : Int → Int =
λ x . {
  case x of
  | 0 => 0
  | 1 => 1
  | n => f (n - 1) + f (n - 2)
  end
} in
f 25

```

Listing 6: A computationally expensive Hazel program with no holes

and evaluation with substitution. If we introduce additional global variables as in the first variation, we observe the behavior in Figure 7.2. The performance of the evaluation with substitution is virtually unchanged, while the performance of evaluation with environments increases. We observe that the increase in evaluation time is not linear. We expect the slowdown to be logarithmic with respect to the number of elements in the environment. On the other hand, if more variables are introduced in an unevaluated branch as in the second variation, then we observe the reversal of the effects on the branches. This is shown in Figure 7.3. The evaluation time when using substitution increases linearly with respect to the length of the unevaluated branch, but evaluation time using environments is roughly unchanged.

These variations show the expected behavior of the two evaluation methods. Often, the performance difference may not be significant, because there is no inherent change in the overall runtime complexity. Substitution eagerly substitutes at binding time, whereas using environments lazily substitutes at lookup time. Introducing additional global variables increases the number of variables in each environment, slowing down evaluation with environments. However, this does not slow down substitution because these variables are never encountered after being bound. On the other hand, substitution necessarily traverses unevaluated branches, whereas evaluation with environment never reaches those branches.

```

let a = 0 in
let b = 0 in
let c = 0 in
let d = 0 in
let e = 0 in
let f : Int → Int =
  λ x . {
    case x of
      | 0 => 0
      | 1 => 1
      | n => f (n - 1) + f (n - 2)
    end
  } in
f 25

```

Listing 7: Adding global bindings to the program in Listing 6

```

let f : Int → Int =
  λ x . {
    case x of
      | 0 => 0
      | 1 => 1
      | n => f (n - 1) + f (n - 2)
      | 0 => f 0 + f 0 + f 0 + f 0 + f 0
    end
  } in
f 25

```

Listing 8: Adding variable substitutions to unused branches to the program in Listing 6

n	Regular	Variables in unused branch					Extra global variables				
		2	4	6	8	10	2	4	6	8	10
22	334	394	509	539	658	677	339	305	302	339	336
23	478	599	695	835	1116	1107	524	442	452	452	455
24	775	929	1214	1332	1518	1686	744	700	729	794	708
25	1233	1502	1874	2310	2398	2723	1171	1189	1134	1104	1231
26	2019	2391	2939	3399	3872	4417	1841	1747	1761	1773	1780

(a) dev branch											
n	Regular	Variables in unused branch					Extra global variables				
		2	4	6	8	10	2	4	6	8	10
22	255	267	276	242	245	243	330	384	417	435	519
23	406	374	376	358	366	330	497	576	573	593	660
24	578	558	559	591	561	569	775	857	911	912	1037
25	851	883	871	864	888	908	1209	1363	1469	1473	1684
26	1318	1388	1382	1398	1399	1415	1935	2262	2302	2356	2492

(b) eval-environment branch											
n	Regular	Variables in unused branch					Extra global variables				
		2	4	6	8	10	2	4	6	8	10
22	255	267	276	242	245	243	330	384	417	435	519
23	406	374	376	358	366	330	497	576	573	593	660
24	578	558	559	591	561	569	775	857	911	912	1037
25	851	883	871	864	888	908	1209	1363	1469	1473	1684
26	1318	1388	1382	1398	1399	1415	1935	2262	2302	2356	2492

Table 7.1: Time (ms) to compute fib(n)

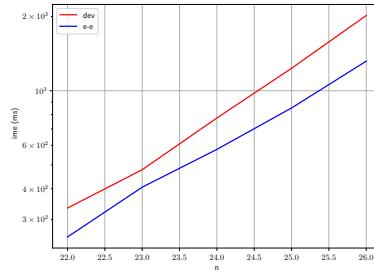
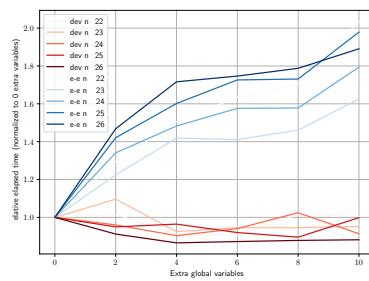
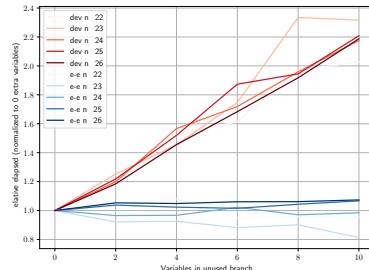


Figure 7.1: Performance of evaluating fib(n)

Figure 7.2: Performance of evaluating $\text{fib}(n)$ with extra global variablesFigure 7.3: Performance of evaluating $\text{fib}(n)$ with an unused branch

7.2 Postprocessing performance

Consider the set of programs described by Listing 4, which motivate the memoization of hole tracking (section 5.3) and the fast structural equality checking algorithm (section 5.5).

We illustrate the performance issues by evaluating the performance issue. The results are shown in tabular form in Table 7.2, and visually in Figure 7.4a. Due to the exponential blowup in elapsed time, we stop recording performance after fifteen consecutive `let` statements.

The exponential performance blowup occurs in three places: in evaluation (due to eager substitution of holes), in hole numbering during postprocessing, and in the structural equality check in `Model.update_program`. All three slowdowns occur due to the lack of memoization of hole closures when traversing a dense graph using a tree traversal algorithm.

We compare the performance to evaluation on the `eval-environment` branch which implements memoization of environments in hole numbering and structural equality. This is shown in Table 7.2, and visually in Figure 7.4b. The evaluation time is greatly improved; total evaluation time remains negligible and never exceeds 7ms. This is the kind of evaluation time a user may expect for an ostensibly simple program.

7.3 FAR performance

We explore two sample edit histories and the effect of FAR on the number of evaluation steps. The current implementation does not look back multiple edit states (i.e., it is a 1-step FAR).

7.3.1 A motivating example

We have an example edit sequence shown in Table 7.3. This shows a possible edit sequence around the motivating example in Listing 5. In it, we have an expensive calculation, and

	dev branch			eval-environment branch		
	Evaluate	Postprocessing	Equality	Evaluate	Postprocessing	Equality
1	0	0	0	0	1	0
2	0	0	0	0	1	0
3	1	2	0	0	1	0
4	1	1	1	1	0	0
5	1	1	2	0	3	0
6	5	1	3	1	0	0
7	4	5	6	2	2	0
8	3	3	14	0	0	0
9	6	18	33	1	0	1
10	14	29	61	0	0	0
11	13	41	91	3	2	0
12	25	145	203	2	0	1
13	65	578	383	2	0	0
14	147	2399	924	1	3	1
15	226	16597	1603	3	0	1
16				1	0	1
17				2	1	1
18				0	3	1
19				0	0	1
20				3	4	0
21				2	0	1
22				0	2	1
23				0	3	1
24				0	6	1
25				1	4	1
26				1	2	1

Table 7.2: Performance of program illustrated in Listing 4

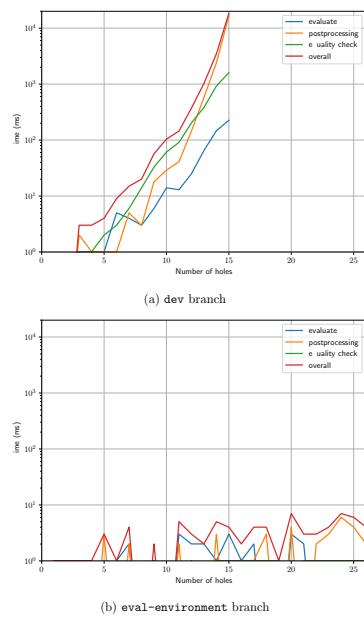


Figure 7.4: Performance of evaluating program in Listing 4

hope to resume the result without redoing the expensive computation. The table shows the sequence of edit states (excluding movement edits). For each edit state, we display the number of steps using regular evaluation, as well as the number of steps for FAR if there is a valid fill operation. We also show the difference in evaluation steps between the FAR evaluation and the regular evaluation (Step Δ), as well as the cumulative difference in evaluation steps. For operations with no valid fill operation, the step difference is zero.

We observe that for the operations before the introduction of the heavy computation $f 25$, there is not a significant difference in the number of steps between normal evaluation and FAR evaluation (when applicable). However, the three steps after the introduction of this computation are valid 1-step FAR operations, and the FAR operation is extremely less expensive. A visualization for this is shown in Figure 7.5. With Figure 7.5a, all edit states after the introduction of the heavy computation also involve the fibonacci calculation. However, in Figure 7.5b, with FAR we see that subsequent calculations roughly only evaluate the filled expression. The cumulative difference in evaluation steps quickly adds up, as evidenced by Table 7.3.

7.3.2 Decreased performance with FAR

Another sample program is shown in Table 7.4. This explores the result of a simpler and perhaps more average program, without an expensive computation. We observe that in this case, the fill operations are typically more expensive than regular operations. This is largely due to the lack of memoization of the re-evaluation of environments in closures, which is described as future work in section 9.2.2. However, the number of evaluation steps are reasonably on the same order of magnitude as normal evaluation.

We note that we only provide evaluation step counts rather than benchmark times for this discussion of FAR. We do not find that benchmarking evaluation is necessary, as it is more or less unchanged from regular evaluation (except for treatment of closures). It may be useful to benchmark the structural diffing algorithm, but this may be more important once a multi-

step FAR is implemented. The current structural diffing with a one-step FAR is a linear pass over the external expression tree, which should have performance characteristics similar to other linear passes over the external expression such as type-checking or elaboration. These algorithms are not a performance bottleneck when compared to naïve algorithms that traverse internal expressions with environments.

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ
<code>let f = ... in</code>	7	-	0	0
<code>let a = ()¹ in</code>				
<code>()²</code>				
<code>let f = ... in</code>	12	21	9	9
<code>let a = f in</code>				
<code>()²</code>				
<code>let f = ... in</code>	17	-	0	9
<code>let a = f ()³ in</code>				
<code>()²</code>				
<code>let f = ... in</code>	58	69	11	20
<code>let a = f 2 in</code>				
<code>()²</code>				
<code>let f = ... in</code>	4762964	-	0	20
<code>let a = f 25 in</code>				
<code>()²</code>				
<code>let f = ... in</code>	4762966	12	-4762954	-4762934
<code>let a = f 25 in</code>				
<code>()² + ()⁴</code>				
<code>let f = ... in</code>	4762966	21	-4762954	-9525879
<code>let a = f 25 in</code>				
<code>()² + 2</code>				
<code>let f = ... in</code>	4792967	13	-4792954	-14288813
<code>let a = f 25 in</code>				
<code>a + 2</code>				

Table 7.3: A program edit history with an expensive computation

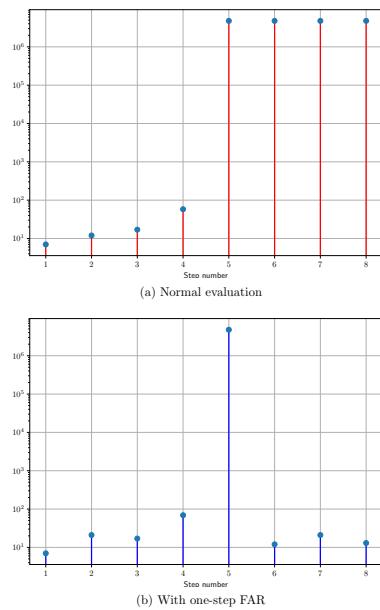


Figure 7.5: Number of evaluation steps per edit in Table 7.3

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ
<code>∅¹</code>	1	-	0	0
<code>let ∅⁶ = ∅⁵ in ∅⁷</code>	2	3	1	1
<code>let x = ∅⁵ in ∅⁷</code>	3	-	0	1
<code>let x = ∅⁵ in let ∅¹² = ∅¹¹ in ∅¹³</code>	4	5	1	2
<code>let x = ∅⁵ in let y = ∅¹¹ in ∅¹³</code>	5	-	0	2
<code>let x = ∅⁵ in let y = 4 in ∅¹³</code>	6	9	3	5
<code>let x = ∅⁵ in let y = 4 in ∅¹³ * ∅¹⁴</code>	8	8	0	5
<code>let x = ∅⁵ in let y = 4 in x * ∅¹⁴</code>	9	14	5	10
<code>let x = ∅⁵ in let y = 4 in x * y</code>	10	11	1	11
<code>let x = 3 in let y = 4 in x * y</code>	11	6	-5	6

Table 7.4: A sample edit history for a simple program

Chapter 8

Discussion of theoretical results

8.1 Expected performance differences between evaluating with substitution versus environments

Section 7.1 discusses the empirical results of a few experiments on the performance of evaluation using the substitution method as originally described and implemented, and evaluation using environments as described in this paper. This shows that the performance gain is highly dependent on the program, and it is not unexpected to see the performance of the substitution model of evaluation outperform the same program evaluated using the environment model. Thus it is difficult to provide a general result about which model is preferred, strictly for performance reasons. Instead, we only provide the highly-parameterized results in the results of the two variations of the fibonacci program.

A way to remedy this is to collect information about expected or average programs, as described in section 9.3. The same is true to determine average performance characteristics of FAR, although this would be related to collecting editing behavior, not characteristics of the program itself.

A non-performance reason may tip the scale in favor of either using the substitution model or the evaluation model. As observed throughout this text, the substitution model

is much simpler to formally state and implement, and may be sufficient in most use cases. However, environments allow for memoization. Environments also allow an implementation to be more amenable to lower-level implementations, because it does not require the runtime to keep a representation of the internal language (whereas substitution does). However, it is also important to keep in mind that environments in this implementation are closely tied to an immutable data structure (due to the frequency of copying and the efficiency of structural sharing), which may frustrate low-level implementations.

8.2 Purity of implementation

The purity of implementation is a recurring theme. While it should not affect the capability of the implementation, there is a strong urge to keep the implementation pure. Elegance, complexity, and runtime overhead is traded off for purity. The main decisions regarding purity are summarized here, and left for the consideration of future implementors.

One offender of performance is the use of the fixpoint form when evaluating recursive functions. This involves extra evaluation steps for unwrapping fixpoints, and can be avoided with self-referential data structures, and more easily implemented using `refs`.

The evaluation process becomes stateful. Every call to `Evaluator.evaluate` takes an `EvalState.t` as both input and output. This maintains state but is still technically pure, much like a state monad. This state includes the environment identifier generator, the fill memoization context, and evaluation statistics such as evaluation steps. This complicates the formalization and the implementation, as we now have additional inputs and outputs to each evaluation judgment. This is also likely less performant than if some global state were used instead. This global state would have to be reset at the beginning of each evaluation or resumed evaluation.

8.3 Summary of metatheorems

Metatheorems 4.1.1 and 4.1.2 establish the type-safety of evaluation with environments. Metatheorems 4.2.1 to 4.3.1 characterize the evaluation boundary and the results of substitution postprocessing. Metatheorem 4.3.2 states that the substitution-postprocessed result of evaluation with environments should match the result when evaluating using substitution. Metatheorem 5.4.1 describes the behavior of the hole closure numbering operation. Metatheorems 6.4.1 to 6.4.2 characterize the static and dynamic correctness of FAR. Metatheorems 6.4.3 to 6.4.4 characterize closures in the outputs of the fill and resume steps of the FAR algorithm.

Most of these theorems describe the expected behavior of the result, such as the conditions on closures in the result. Thus we may think of these metatheorems as “checksum theorems” to informally check the correctness of the work. The correctness of these metatheorems is informally justified using by considering relevant inference rules.

Additionally, we may define “adequacy theorems,” which state that the result of the implemented algorithm matches the inference rules, and implement a series of checks in the code to test that this is indeed the case. However, Hazel does not have such a testing framework in place, and a more rigorous test framework is left for future work.

8.4 FAR for notebook-style editing

One of the primary qualities of computational notebooks is the ability to run sections of the code at a time, primarily for computational purposes. We may reproduce this behavior in a limited way. This is an extension of the discussion from the introduction of Hazelnut Live [2].

We may model a notebook-style program as a linear sequence of sections or cells. Each cell, can be modeled as a set of variable bindings: the “outputs” of evaluating the section. We may interpret a `let` statement in Hazel to be a single-output section.

```

%% section 1
x = 5;
y = 4;
%% section 2
z = 3 * x + y;
x = 5 * z;
%% section 3
% a = ...

```

(a) Sample notebook-style program in MATLAB

```

let x = 5 in
let y = 4 in
let z = 3 * x + y in
let x = 5 * z in
()
```

(b) Sample notebook-style program in Hazel

Figure 8.1: Comparison of notebook-style programs in MATLAB and Hazel

Consider the sample notebook-style program shown in Figure 8.1a. Here, we have two sections, which compute three different variables. If we add a third section containing the statement `a = x + y;`, then we may obtain the value of `a` by only executing the third section and not re-evaluating the first two sections, because the workspace is saved.

Now, consider the comparable program in Figure 8.1b. If the user types in the expression `x + y` or `let a = x + y in ()`, then a fill operation is detected, and the new expression is computed with the environment stored in hole 1's closure. In fact, if the user continues to make any changes below the fourth `let` expression, they will all be considered to be valid n -step fill operations against the edit state shown in Figure 8.1b, so the first four statements never have to be re-evaluated. As the user continues to edit downwards, new holes will be created and the newest edits may be considered fill operations from more recent edits.

Additionally, we have the benefit of reproducibility not present in MATLAB. In most notebook-style editors, such as MATLAB's live editor, running sections out of order may cause a different program output than if the program had been run in order. For example, if the user runs the section section twice after evaluating the first section Figure 8.1a is run twice in a row, then the output would be different than if the program was run from start to finish. On the other hand, the Hazel program in Figure 8.1b will always give the same result as if evaluation had begun from the top, as guaranteed by the commutativity property of FAR.

The limitation of using Hazel as a computational notebook is that the fill-and-resume operation is limited to cases where there is a hole in a previous edit state as a parent of the root of the diff. If a user programs by always appending to or editing near the bottom of their program, FAR will be very beneficial because most edits will lead to valid fill operations. It may be difficult to quantify the real performance benefit due to large variations in programming styles. Different heuristics for choosing a past edit state to compare, as discussed in section 9.2.3, will also affect the performance of FAR.

8.5 Summary of generalized concepts

The work performed for FAR leads us to the following nice generalizations of some of the concepts we've encountered through this work.

8.5.1 Generalized closures and the evaluation boundary

Generalized closures form an integral part of this implementation. In the literature, the term “closure” usually refers to function closures, but we find that allowing for a general container expression with a bound environment is extremely useful for our implementation. Not only do they tidy up the implementation by “factoring out” environments from the numerous expression forms that require them¹, but they characterize the evaluation boundary: expressions in the evaluated result that exist within a closure are “paused” evaluations that may be resumed later. Separating holes from closures also helps to facilitate fill-and-resume because we wish to substitute the hole with d_{full} without discarding the environment. In summary, closures are harmonious with Hazel’s live environment characteristics.

¹This is true even in the case of evaluation with substitution, by separating environments from hole closures.

8.5.2 A generalization of non-empty holes

Non-empty holes play a central role in Hazel's ability to provide continuous feedback, as well as in the ability to resume computation in FAR. We may interpret an empty hole as encapsulating a well-formed expression in some incompatible outer expression.

It may be helpful to also imagine that the entire program lies in a non-empty hole. In this interpretation, regular program evaluation (from the start) may be considered a degenerate case of fill-and-resume, where the root of the diff is a non-fill diff that gets propagated up to this non-empty hole. This hole will also nicely serve as the parent for all holes in a non-complete program in the `HoleClosureInfo.t`, although the parent of this hole would then not be well-defined.

8.5.3 FAR as a generalization of evaluation

It is possible to express every evaluation operation as a n -step FAR operation, assuming that we have the ability to look back an arbitrary number of edit states. The intuition behind this is that the initial state of a program is the empty hole $\langle \rangle^1$. It is trivial to prove that using the rules given by the fill diff that every edit state produces either a no diff or a fill diff judgment against this edit state.

For practical reasons, it may be less efficient to perform a fill diff against an arbitrary number of states, or even to store the entire history of a program. Also, the whole history of a program may not be available. In the case of a parsed program or an example program, which is specified as an edit state rather than a history of edit actions, we would need additional machinery to produce a possible series of edit actions that leads to this state from the empty state.

Chapter 9

Future work

9.1 Simplification of the postprocessing algorithms

The postprocessing algorithms presented in sections 4.3 and 5.4.1 are both memoized to avoid repeatedly postprocessing an environment multiple times. However, the latter algorithm (hole numbering) complicates memoization. This is due to the fact that holes need to be re-postprocessed each time it is encountered in a new parent in order to track the closure parents.

It would be desirable if we can avoid working around this, perhaps by reformulating the way hole parents are tracked. A possible solution is to implementing hole parent tracking as a separate postprocessing pass after hole closure numbering.

9.2 Improvements to FAR

9.2.1 Finishing the implementation of FAR

The implementation of FAR completed for this thesis is a limited proof-of-concept. The FAR detection mechanism (the structural diffing algorithm) and preprocessing steps are complete. However, due to limited time constraints, the implementation has not achieved parity with

the theoretical exploration of FAR explored in this paper.

Firstly, a list of past edit states should be made accessible to the FAR detection algorithm. The current structure of the Hazel toplevel and the undo history makes this trickier because there may be multiple unrelated histories caused by switching programs (“cards”) in the top panel of the Hazel UI.

Secondly, the evaluation result from FAR should be memoized alongside the results from regular evaluation. The current implementation of normal evaluation is memoized and is not aware of the FAR operation.

Thirdly, the result of evaluating the program (`Result.t`) is not currently stored in the model, but the program’s edit state (`Program.t`) is. This means that when the program result is needed in the Hazel UI, the program is re-evaluated. This is usually not a problem because of memoization, but FAR results are not memoized. It would be better to store the results of evaluation in the model and undo history to avoid this trouble.

Fourthly, is a slight issue with the current description of the `FillExp` variant $(\text{d})_i$. This requires us to have access to the hole closure identifier i , which comes from the postprocessed result. However, we perform the FAR operation on the un-postprocessed result. To remedy this, we may begin evaluation from the post-processed result, which may cause us to change some of our assumptions about the evaluation boundary. Another solution would be to have an alternative postprocessing operation that renames holes but leaves alone the evaluation boundary.

9.2.2 Memoization for environments during re-evaluation

We introduced fill expressions $(\text{d})_i$ in section 6.2.2 in order to memoize the evaluation of filled expressions during re-evaluation.

It will also be beneficial to memoize the re-evaluation of closure environments. To illustrate why this is the case, consider the case of Figure 6.15. In this example, the environment is evaluated twice, even though it is the same physical environment. Each environment will

be re-evaluated each time it is encountered, leading to the same exponential blowup problem encountered when dealing with postprocessing in Listing 4. The implementation of this memoization is the same as before; we will need a new environment state variable mapping environment identifiers to evaluated environments.

9.2.3 Choosing the edit state to fill from

There are a number of possible design decisions when searching for a valid hole fill. Firstly, one must decide the maximum number of edit states to search: should it be a fixed number of edit states, or should it be given a fixed time budget? Is it best to cache edit states that recently led to a fill operation (à la LRU cache)? Is the most recent edit state that leads to a valid fill usually the best candidate, or even a good candidate? Would it be best to allow for user-configurable settings, or perhaps even for the user to manually select the previous edit from which to fill?

It will be useful to collect empirical data about user-editing behaviors, as discussed in section 9.3, in order to better tune these parameters. Alternatively, we may allow for some user configuration of FAR, as discussed in section 9.2.4.

9.2.4 User-configurable FAR

The FAR procedure as described is a completely automatic process. However, the choice of which past edit state to choose may be tricky, as described in section 9.2.3. It may be desirable for the user to manually set a past edit state as an “anchor” from which to set FAR, and thus override an automatic choice of past edit state. This reifies the concept of sections or section breaks from computational notebooks. However, since a user-chosen edit state may not lead to a valid fill diff, edit states that lead to an invalid fill diff will still have to be evaluated from scratch.

It may also be desirable to disable FAR completely, whether for debugging purposes or because it does not help performance much in the given circumstance. This should be a

toggable parameter in the sidebar.

Any changes to the interactiveness of the FAR operation will have a great effect on intuitiveness, ease-of-use, editing speed, and ostensible performance. These will all play a great effect on the usability and viscosity of editing a Hazel program.

9.2.5 UI changes for notebook-style evaluation

We have motivated the use of Hazel with FAR for notebook-style evaluation in section 8.4. It may be useful to the programmer to have additional visual aids to help enforce the benefits of resumed evaluation. This is currently a being developed on the `lab-notebook-ui` branch of the GitHub repository.

It may be useful to format the code editor using a series of sections. These sections will be syntactic sugar for sequences of let statements, and edits to the code in a section will only cause the current and subsequent sections to re-evaluate. This also presents a method to choose the edit state from which to fill that is intuitive to the user.

Another useful feature to evaluate FAR is to show evaluation statistics in the UI. The two evaluation statistics described in section 6.6.1 may be useful information to the user. These may inform the user what types of operations are more expensive when FAR is taken into account, and may actually promote a style of editing that maximizes the usage of FAR.

9.3 Collection of editing statistics

The effects of switching evaluation to use environments instead of substitution, and the effect of different methods of choosing a previous edit state for FAR may drastically change the expected performance gain. We may only quantify expected differences in performance when concerned with specific types of programs or editing patterns. We also do not know the effect of manual or automatic FAR on editing viscosity.

Thus it will be useful to collect empirical statistics about user editing patterns, in order

to gain a useful insight into the general expected effect on performance of this work. This information will likely help inform many other Hazel subprojects. Since Hazel is hosted online and is accessible to all, collecting usage statistics is technically very easy. Other online programming language environments, such as the Hedy incremental programming environment for programming education [36, 37], have collected user editing statistics to help direct their work.

9.4 Generalization of memoized methods

Memoization plays an important part of this work. Memoization of environment plays a role in the substitution postprocessing (section 4.3), hole numbering postprocessing (section 5.4.1), structural equality checking (section 5.5), evaluation of filled expressions (section 6.2.3), and re-evaluation of environments (section 9.2.2). Despite this, each time memoization is used, the implementation and formalization are *ad hoc*. This is due to the scattered discovery of performance problems and realization that memoization would be useful.

A future implementation of memoized algorithms will be well-served by a generalized characterization of memoization by environments. This should include a generalized notation, so that new memoized algorithms can be easily introduced. This should also include a precise characterization of what algorithms may be memoized; for example, the difficulty presented in section 9.1 is due to the difficulty of memoizing the hole parent tracking process as part of the hole numbering process.

9.5 Mechanization of metatheorems and rules

The metatheory of Hazelnut and Hazelnut Live were proved using the Agda proof assistant [9, 10]. A similar proof of the metatheory presented in this work is left to future work. Due to time constraints, we are satisfied with using the intuition presented as justifications for the metatheorems in this thesis.

Chapter 10

Conclusions and recommendations

This thesis explores several practical advancements to the dynamic semantics of Hazel, an implementation of a live programming editor with typed holes. We develop rules and a metatheory for the proposed changes as well as a working implementation for most of the rules provided.

The first proposed change is the switch from using variable substitution at binding time to looking up variables in an environment at runtime. This implementation leads to the introduction of closures to the internal language of Hazel, as well as a postprocessing step to restore the same result that would have been achieved by evaluation using the substitution model. Initially, we begin with the conventional function closures and the hole closures introduced in Hazelnut. Later, we introduce generalized closures, which subsume function and hole closures, and represent any paused evaluation. Generalized closures play a critical role in the fill-and-resume (FAR) optimization.

The second major change is the use of the environment model to memoize operations that occur on shared environments. Environments are uniquely numbered to allow for lookup and memoization. Memoization is applied to the hole renumbering process (in the process changing the useful interpretation from hole instances to hole closures), and to speed up the structural equality checking. Memoization may also be helpful with the re-evaluation with

filled expressions and environments during the FAR re-evaluation process, but this has not been fully implemented yet.

The third major contribution is a set of practical considerations to implementing FAR, as originally proposed in Hazelnut Live. FAR promotes resuming evaluation from a previous evaluation result when an edit action is performed, as opposed to restarting evaluation from the beginning on every edit action. Firstly, we describe a structural diffing algorithm for detection of a valid fill operation. This algorithm is intuitive and robust to work between an arbitrary past edit state and the current edit state (a n -step fill operation). We also provide the basic semantics for the fill (preprocessing) and resume (re-evaluation) operations. A 1-step FAR operation is implemented as a proof of concept.

We evaluate the performance of these methods empirically, via a series of benchmarks of sample programs. We compare the difference between the current main development branch on the `dev` branch to an updated evaluation model implementing the changes proposed in this thesis, in the `eval-environment` and `fill-and-resume-backend` branches. The results qualitatively match the expectation. Evaluation with environments is beneficial for performance when lazy variable lookups are reduced and the environment size is small. Substitution may be beneficial for performance when the number of substitution passes is small. The memoization of environments solves the performance issue in the program that motivated the memoization of environments in the hole numbering and structural equality checking processes. FAR provides a great improvement in efficiency when there is a valid fill operation and expensive re-computations can be avoided. However, FAR may sometimes be more expensive than regular evaluation from the beginning, due to the recursive re-evaluation of environments. Future work on this environment memoization and the implementation of the n -step fill operation should further improve the performance benefit of FAR.

We do not prove the correctness of the implementation. We instead provide a metatheory governing the implementation and provide a logical intuition for the correctness of the proposed metatheorems. We leave formal proofs of the metatheory and proofs of the com-

pleteenes of the metatheorems to future work.

The primary goal of this work is to inform future development on Hazel or related research efforts, and the explanations and motivations have been written in enough detail to allow for others to independently reproduce this work. The implementation of the rules in this work are intended to act as a reference implementation and not necessarily be incorporated directly into the main development branches; the theoretical discussion is the more useful part of this work. We discuss practical tradeoffs of implementing evaluation with environments. Evaluation using environments leads to some improvement in performance in many programs where lazy variable lookup is beneficial, and leads to a major improvement in performance in some programs due to the effect of memoizing environments, but comes at the expense of a great deal of extra complexity that may obscure Hazel's original goal in approaching the gap problem. We also describe a possible implementation of FAR and entrypoint for the FAR algorithm, with possibilities for further memoizing re-evaluation. The empirical results that are presented may serve as a guideline for performance benefits, but it will be useful to collect user editing and program statistics to better evaluate the average or expected performance benefit of the proposed changes. Along the way, we introduce several novel generalizations that both simplify implementation and give nice theoretical interpretations, such as generalized closures and the presentation of n -step FAR as a generalization of evaluation.

References

- [1] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.
- [2] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *PACMPL*, 3(POPL), 2019.
- [3] hazelgrove. Hazel dev branch live demonstration. <https://hazel.org/build/dev/>, Mar 2022.
- [4] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.
- [5] Eyal Lotem and Yair Chuchem. Lamdu. <https://www.lamdu.org/>.
- [6] Markus Voelter, Daniel Ratiu, Bernhard Schaez, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140, 2012.
- [7] hazelgrove. hazelgrove/hazel. <https://github.com/hazelgrove/hazel>.

- [8] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda-a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.
- [9] hazelgrove. hazelgrove/agda-popl17. <https://github.com/hazelgrove/agda-popl17>.
- [10] hazelgrove. hazelgrove/hazelnut-dynamics-agda.
- [11] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):1–38, jun 2022.
- [12] Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 71–78, 2005.
- [13] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.
- [14] Adam Michael Gundry. *Type inference, Haskell and dependent types*. PhD thesis, University of Strathclyde, 2013.
- [15] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.
- [16] Gilles Kahn. Natural semantics. In *STACS*, 1987.
- [17] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.
- [18] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.
- [19] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. Real world ocaml. 2014.

- [20] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.
- [21] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [22] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [23] Robert Harper and Christopher A Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction*, pages 341–388, 2000.
- [24] Peter Sestoft. Runtime code generation with JVM and CLR. Available at <http://www.dina.dk/sestoft/publications.html>, 2002.
- [25] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical computer science*, 1(2):125–159, 1975.
- [26] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.
- [27] Hannah Potter and Cyrus Omar. Hazel tutor: Guiding novices through type-driven development strategies. *Human Aspects of Types and Reasoning Assistants*. <https://hazel.org/hazeltutorhatra2020.pdf>, 2020.
- [28] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
- [29] David Moon. hazelgrove/tylr. <https://github.com/hazelgrove/tylr.git>.

- [30] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDermid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. *SIGPLAN Not.*, 48(6):95–104, jun 2013.
- [31] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in science & engineering*, 9(3):21–29, 2007.
- [32] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11. IEEE, 2020.
- [33] Jérôme Vouillon and Vincent Balat. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.
- [34] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 511–525, 2021.
- [35] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–49, 2008.
- [36] Felienne Hermans. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*, pages 259–270, 2020.
- [37] Marleen Gilsing and Felienne Hermans. Gradual programming in Hedy: A first user study. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE, 2021.

Appendix A

Primer to the λ -calculus

This appendix acts as a self-contained primer on the λ -calculus, which serves as the theoretical foundation for the Hazel programming language and many other functional programming languages. Much of this material comes from a standard introductory text in programming languages, such as [18]. While this thesis focuses on the dynamic semantics, we do not attempt to separate it from the grammar and static semantics of the λ -calculus, for they are intimately connected.

A.1 The untyped λ -calculus

Church introduced the *untyped λ -calculus* Λ as an example of a simple universal language of computable functions, and it forms the foundation for the syntax and evaluation semantics of functional programming languages.

The grammar of Λ is very simple, only comprising three forms (excluding parentheses¹), shown in Figure A.1.

¹The imperative programmer with a background in a C-family language be warned: parentheses are not required for function application. Rather, space (\cdot) is an infix operator that represents function application in Λ and many functional languages. It traditionally is left-associative and has the highest precedence of any infix operator. Parentheses around function arguments are only required when it affects the order of operations. An exception to this rule in functional programming is in the LISP family of languages, in which parentheses specify function application rather than operator precedence, and space separates operators and operands, but that is not the interpretation here.

$e ::= x$	(variable)
$ \lambda x.e$	(λ -abstraction)
$ e e$	(function application)

Figure A.1: Grammar of Λ

$\frac{}{\lambda x.e \Downarrow \lambda x.e} \text{ A-ELam}$	$\frac{e_1 \Downarrow \lambda x.e'_1 \quad [e_2/x]e'_1 \Downarrow e}{e_1 e_2 \Downarrow e} \text{ A-EAp}$
--------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------

Figure A.2: Dynamic semantics for Λ

The static semantics of this syntax are very simple: every expression in Λ is well-formed iff all variables are bound².

The dynamic semantics are similarly simple, shown in Figure A.2 using a big-step semantics. λ -abstractions are values (expressions that evaluate to themselves), and application is applied by substituting variables³.

Λ is an example of a Turing-complete language. One of the key characteristics to such a language is the ability to implement recursive algorithms. To implement recursion, a function must be able to refer to itself. Since there is no construct to bind an expression to a variable other than λ -abstractions (i.e., there is no construct such as OCaml's `let rec` expressions or other standalone variable declarations), one must pass a self-reference of a function to itself. For example, let us consider the example of a factorial function in Λ ⁴.

$$\text{fact}^* \equiv \lambda f. \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1)$$

²There are no typing rules in the static semantics, because there is only a single type: the recursive arrow type $\tau ::= \tau \rightarrow \tau$. Thus, it may be more correct to say that Λ is “un-typed” as opposed to “untyped,” as noted in [18]. Thus no type errors will occur when evaluating a (well-formed) expression in Λ .

³The substitution of the function variable during function application is known as β -reduction. Renaming of bound variables (a process known as α -conversion) is used to avoid substituting variables of the same name bound by a different binder.

⁴For sake of illustration, the language is extended with a conditional statement, integers, and simple integer operators.

$\tau ::= \tau \rightarrow \tau$	(function type)
b	(base type)
$e ::= c$	(constant)
x	(variable)
$\lambda x : \tau.e$	(type-annotated function)
$e e$	(function application)
$\text{fix } f : \tau.e$	(fixpoint)

Figure A.3: Syntax of Λ_{\rightarrow}

To facilitate the recursion, we need the help of an auxiliary operator which converts a recursive function formulated with a self-reference parameter as shown above. The Y-combinator is such an operator. The operation of this operator is made clear by working through the β -reduction of the `fact` function.

$$Y \equiv \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$

fact $\equiv Y\ \text{fact}'$

A more thorough discussion of Λ and the Y-combinator is left to standard material on programming language theory, such as [18].

A.2 The simply-typed λ -calculus

While the λ -calculus is Turing complete and sufficient to represent any computation, it is not practical in terms of efficiency or usability if all data is represented with functions⁵.

The *simply-typed λ -calculus* (STLC) Λ_{\rightarrow} extends Λ with one or more base types b_i , such as integers, booleans, or floating-point numbers. Consider the case of a single base type b . The extended grammar is shown in Figure A.3.

⁵All data may be represented in terms of functions with a notation called the Church encoding. For example, there are standard Church encodings for natural numbers, for boolean values and conditionals, and for pairs (`cons`), which can be used to construct structured data.

$$\begin{array}{c}
 \frac{}{\Gamma \vdash c : b} \Lambda_{\rightarrow}\text{-TConst} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau} \Lambda_{\rightarrow}\text{-TVar} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.e) : \tau_2} \Lambda_{\rightarrow}\text{-TLam} \\
 \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau} \Lambda_{\rightarrow}\text{-TAp} \quad \frac{}{\Gamma \vdash (\text{fix } f : \tau.e) : \tau} \Lambda_{\rightarrow}\text{-TFix}
 \end{array}$$

Figure A.4: Static semantics of Λ_{\rightarrow}

$$\begin{array}{c}
 \frac{}{c \Downarrow c} \Lambda_{\rightarrow}\text{-EConst} \quad \frac{\lambda x : \tau.e \Downarrow \lambda x : \tau.e}{\lambda x : \tau.e \Downarrow \lambda x : \tau.e} \Lambda_{\rightarrow}\text{-ELam} \\
 \frac{e_1 \Downarrow \lambda x : \tau.e'_1 \quad [e_2/x]e'_1 \Downarrow e}{e_1 e_2 \Downarrow e} \Lambda_{\rightarrow}\text{-EAp} \quad \frac{[\text{fix } f : \tau.e/f]e \Downarrow e'}{\text{fix } f : \tau.e \Downarrow e'} \Lambda_{\rightarrow}\text{-EFix}
 \end{array}$$

Figure A.5: Dynamic semantics of Λ_{\rightarrow}

The grammar is extended to include constants of the base type. The type of functions parameters must be annotated⁶.

We now define what it means for a program in Λ_{\rightarrow} to be well-typed. The typing judgments shown in Figure A.4 assign a type to a Λ_{\rightarrow} program.

The dynamic semantics are not much different than Λ . Additional evaluation rules are defined for constants and fixpoints; evaluation of λ -abstractions and function application remains the same. The dynamic semantics are shown in Figure A.5.

We may characterize type systems by establishing certain desirable properties. One such property is *soundness*. Soundness means that if a program in Λ_{\rightarrow} type-checks, then it will not fail with a type error at run-time. This property is not necessary to prove for Λ because there is only one type in Λ , the recursive type $\tau ::= \tau \rightarrow \tau$.

There is an additional expression form in Λ_{\rightarrow} . This is the *fixpoint form*, $\text{fix } f : \tau.e$. The fixpoint is a primitive operator with the same purpose and evaluation behavior as the

⁶This is in the simplest case of type-assignment. With a type inference system such as bidirectional typing as described in Section 2.1.3, some type annotations may be optional.

Y -combinator: it allows for self-reference, and thus general recursion. The reason for the explicit fixpoint operator is that the Y -combinator is ill-typed. Self-reference is inherently poorly-typed and requires a primitive operator, since it involves a function which takes itself as a parameter (leading to an infinitely-recursive arrow type). With the fix operator, we may express the factorial function as shown below⁷.

```
fact ≡ fix f : int → int.λx : int.if x = 0 then 1 else x * f(x - 1)
```

The fixpoint operator is introduced in Plotkin's System PCF [18], and is used to implement recursion in Hazel's evaluator, which uses a substitution-based evaluation.

Λ_{\rightarrow} is a practical foundation for many functional languages. Standard exercises include extending Λ_{\rightarrow} with multiple base types (such as integers and booleans), conditional expressions, `let`-expressions, and `case`-expressions. The basic type system can be extended to use type inference algorithms or support more advanced types.

A.3 The gradually-typed λ -calculus

We have discussed Λ_{\rightarrow} , which involves a simple *static typing* system, as type checks are part of the static semantics. However, we may extend Λ with an additional base type but without a static semantics. In this case, a well-formed expression may fail at run-time due to type errors – thus, types are checked in the dynamic semantics and this is known as *dynamic typing*. The benefit of static typing is soundness and performance (as run-time type checks are relatively slow). The benefit of dynamic checking is to avoid annotating types⁸, and thus more quickly prototype or refactor programs.

A hybrid approach is the *gradually-typed λ -calculus* $\Lambda_{\rightarrow}^?$, originally proposed by Siek

⁷In this example, we assume that the base type $b \equiv \text{int}$, and that conditionals and primitive integer operations extend Λ_{\rightarrow} .

⁸Note that type inference systems in a statically-typed system also allow for reduced type-annotations, but may still require some annotations when not enough information is given for type inference.

$\Gamma \vdash e_1 : \tau_1$	$\tau_1 \blacktriangleright \tau_2 \rightarrow \tau_3$	$\Gamma \vdash e_2 : \tau_3$	$\Lambda_{\rightarrow}^2.TAp$	$\frac{\Gamma \vdash e : \tau \quad \tau \sim \tau'}{\Gamma \vdash e : \tau'}$	$\Lambda_{\rightarrow}^2.TSub$
<hr/>					

Figure A.6: Updated static semantics of Λ_{\rightarrow}^2

and Taha [20, 21]⁹. In Λ_{\rightarrow}^2 , all type annotations are optional and offer a “pay-as-you-go” benefit. A completely unannotated Λ_{\rightarrow}^2 program acts like dynamic typing (Λ extended with base type(s) but no static semantics), with run-time casts and the possibility of run-time type failures. A completely annotated Λ_{\rightarrow}^2 program is equivalent to a Λ_{\rightarrow} program. The performance cost of run-time casts and the possibility of run-time type failures only occurs when evaluating expressions with unannotated terms.

The grammar of Λ_{\rightarrow}^2 is almost exactly the same as Λ_{\rightarrow} , except that we add a new type $*$, indicating an unspecified type. Now, λ -abstractions annotated with this type may be considered to be unannotated. We define the notation $\lambda x.e \equiv \lambda x : * . e$.

The static semantics of Λ_{\rightarrow}^2 , shown in Figure A.6, is expectedly also similar to Λ_{\rightarrow} . The only rule that differs is the rule for function application. We also write a new rule for subsumption, which states that if $\Gamma \vdash e : \tau$, then e may also be assigned any consistent type.

Two new judgments are introduced here. The first is the *matched arrow judgment* $\tau_1^- \blacktriangleright_{\rightarrow} (\tau_2 \rightarrow \tau_3)^+$, which is a notational convenience which allows us to write a single rule for arrow types, which may either be a hole or an arrow type. This judgment is defined by the rules in Figure A.7.

The second new judgment is the *type consistency judgment* $\tau_1^- \sim \tau_2^-$. This judgment defines the typing relation of the unknown type to other types: every type is consistent to the hole type. Thus any type will type-check where a hole is expected, and vice versa. This relation is reflexive, symmetric, and non-transitive¹⁰. The rules for type consistency

⁹The material presented in this section originates from Siek et al. [20, 21], but the notation conventions follow from Hazelnut Live [2] in order to stay consistent with the rest of this paper. The symbol for the hole type $*$ originates from [21]. The cast calculus notation is an improved notation introduced in [21] and also used in [2].

¹⁰It may seem unintuitive at first that type consistency is a symmetric relationship, because it may seem

$$\boxed{\frac{\tau \triangleright \tau_1 \rightarrow \tau_2}{\tau \text{ has matched arrow type } \tau_1 \rightarrow \tau_2} \quad \frac{\begin{array}{c} * \triangleright \tau_1 \rightarrow \tau_2 \\ * \triangleright \tau_2 \end{array}}{* \triangleright \tau_1 \rightarrow \tau_2} \text{MAHole} \quad \frac{\tau_1 \rightarrow \tau_2 \triangleright \tau_1 \rightarrow \tau_2}{\tau_1 \rightarrow \tau_2 \triangleright \tau_1 \rightarrow \tau_2} \text{MAArr}}$$

Figure A.7: Matched arrow judgment

$$\boxed{\frac{\tau_1 \sim \tau_2}{\tau_1 \text{ is consistent with } \tau_2} \quad \frac{\begin{array}{c} * \sim \tau \\ \tau \sim * \end{array}}{\tau \sim \tau} \text{TCHoleTyp} \quad \frac{}{\tau \sim *} \text{TCTypHole} \quad \frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2}{(\tau_1 \rightarrow \tau_2) \sim (\tau'_1 \rightarrow \tau'_2)} \text{TCArr}}$$

Figure A.8: Type consistency judgment

are shown in Figure A.8.

Evaluation of a gradually-typed language introduces a cast calculus, which performs run-time type checking. To do this, we design another language, $\Lambda_{\rightarrow}^{(r)}$, whose grammar is identical to $\Lambda_{\rightarrow}^?$, except for the introduction of a new expression form indicating a run-time cast, $d(\tau \Rightarrow \tau')$. We also define the notation $d(\tau \Rightarrow \tau' \Rightarrow \tau'') \equiv d(\tau \Rightarrow \tau')(\tau' \Rightarrow \tau'')$. We refer to $\Lambda_{\rightarrow}^?$ as the *external language* and $\Lambda_{\rightarrow}^{(r)}$ as the *internal language*. We denote expressions in $\Lambda_{\rightarrow}^?$ with the letter e and expressions in $\Lambda_{\rightarrow}^{(r)}$ with the letter d . We only define static semantics on $\Lambda_{\rightarrow}^?$ and only define dynamic semantics on $\Lambda_{\rightarrow}^{(r)}$. The process of converting from the external language to the internal language (i.e., the process of cast-insertion) is called *elaboration*.

Usually, elaboration includes the type-checking operation rather than being a separate operation. Elaboration fails if type checking fails. A preservation theorem may be stated that the type assigned by elaboration is the same as the type assigned by the type assignment judgment.

The elaboration process is governed by the judgment $\Gamma^- \vdash e^- \rightsquigarrow d^+ : \tau^+$. For most

more like a subtyping relation. However, a major revolution in Siek and Taha's original formulation of $\Lambda_{\rightarrow}^?$ is that the symmetric subtyping relation is more suitable than the subtyping relations that had been explored in earlier works such as Thatté's quasi-static typing [20].

$\boxed{\Gamma \vdash e \rightsquigarrow d : \tau}$	e elaborates to d of type τ given typing context Γ
$\Gamma \vdash c \rightsquigarrow c : b$	$\Lambda_{\rightarrow}^{\tau}\text{-ElConst}$
$\Gamma, x : \tau_1 \vdash e : \tau_2$	$\Gamma \vdash \lambda x : \tau_1. e \rightsquigarrow (\lambda x : \tau_1. e) : \tau_1 \rightarrow \tau_2$
	$\Lambda_{\rightarrow}^{\tau}\text{-ElLam}$
	$\Gamma \vdash \text{fix } f : \tau. e \rightsquigarrow (\text{fix } f : \tau. e) : \tau$
	$\Lambda_{\rightarrow}^{\tau}\text{-ElFix}$
$\Gamma \vdash e_1 \rightsquigarrow d_1 : \tau_1$	$\tau_1 \blacktriangleright \tau_3 \rightarrow \tau_4$
$\Gamma \vdash e_2 \rightsquigarrow d_2 : \tau_2$	$\tau_2 \rightsquigarrow \tau_3$
	$\Gamma \vdash e_1 e_2 \rightsquigarrow (d_1(\tau_1 \Rightarrow \tau_3 \rightarrow \tau_4)) (d_2(\tau_2 \Rightarrow \tau_5)) : \tau_4$
	$\Lambda_{\rightarrow}^{\tau}\text{-ElApp}$

Figure A.9: Elaboration in $\Lambda_{\rightarrow}^{\tau}$

expression types, the expression in the external language elaborates to itself. The only exception is function applications, in which dynamic casts are inserted. The elaboration process is described in Figure A.9.

We may now define a dynamic semantics on $\Lambda_{\rightarrow}^{(\tau)}$. The following dynamic semantics is simplified from Sick et al.’s formulation for the sake of clarity¹¹ and is not intended to be a precise description of the evaluation semantics. As before, the dynamic semantics are described using a big-step notation, where the judgment $d \rightsquigarrow \text{final}$ indicates that d is a value¹². There is also the possibility of a dynamic cast error, given by rule $\Lambda_{\rightarrow}^{(\tau)}\text{-ECastFail}$. The dynamic semantics is shown in Figure A.10.

Hazel’s core calculus heavily borrows from $\Lambda_{\rightarrow}^{\tau}$ and $\Lambda_{\rightarrow}^{(\tau)}$. The rules for casts will remain unchanged for the dynamic semantics when switching to use the environment mode of evaluation.

¹¹Sick [21] introduces the idea of *ground types* and the *matched-ground judgment*. Casts can only succeed or fail between ground types. Additionally, Sick et al. describes *blames* and *frames* to encapsulate errors. Ground types are carried over to Hazel’s formulation [2], but blames and frames are not currently implemented in Hazel.

¹²The small-step semantics is perhaps more clear here, as it more clearly illustrates the isolated effect of the cast operation. Both Sick et al. [20, 21] and Hazel’s formulation [2] describe the dynamic semantics using a small-step semantics. Hazel adds the concept of the *final judgment* to delineate values. However, we use a big-step semantics to remain consistent with the rest of the notation in this work.

$$\begin{array}{c}
\frac{}{c \text{ final}} \Lambda_{\rightarrow}^{(\tau)} \text{-VConst} \quad \frac{}{\lambda x : \tau. d \text{ final}} \Lambda_{\rightarrow}^{(\tau)} \text{-VLam} \quad \frac{}{e(\tau \Rightarrow *) \text{ final}} \Lambda_{\rightarrow}^{(\tau)} \text{-VBoxedVal} \\
\frac{d \text{ final}}{d \Downarrow d} \Lambda_{\rightarrow}^{(\tau)} \text{-EVal} \quad \frac{[\text{fix } f : \tau. d/f] d \Downarrow d'}{\text{fix } f : \tau. d \Downarrow d'} \Lambda_{\rightarrow}^{(\tau)} \text{-EFix} \\
\frac{d_1 \Downarrow \lambda x : \tau. d'_1 \quad [d_2/x] d_1 \Downarrow d}{d_1 \ d_2 \Downarrow d} \Lambda_{\rightarrow}^{(\tau)} \text{-EAp} \\
\frac{d_1 \Downarrow d' (\tau_1 \rightarrow \tau_2 \Rightarrow \tau'_1 \rightarrow \tau'_2) \quad (d'_1 (d_2 (\tau'_1 \Rightarrow \tau_1)) (\tau_2 \Rightarrow \tau'_2) \Downarrow d)}{d_1 \ d_2 \Downarrow d} \Lambda_{\rightarrow}^{(\tau)} \text{-ECastAp} \\
\frac{d \Downarrow d'}{d(\tau \Rightarrow *) \Downarrow d'} \Lambda_{\rightarrow}^{(\tau)} \text{-ECastId} \quad \frac{d \Downarrow d'}{d(\tau \Rightarrow * \Rightarrow \tau) \Downarrow d'} \Lambda_{\rightarrow}^{(\tau)} \text{-ECastSucceed} \\
\frac{}{d(\tau \Rightarrow * \Rightarrow \tau') \text{ castfail}} \Lambda_{\rightarrow}^{(\tau)} \text{-ECastFail}
\end{array}$$

Figure A.10: Dynamic semantics of $\Lambda_{\rightarrow}^{(\tau)}$

Appendix B

Reproduced rules from Hazelnut Live

B.1 Hazelnut Live dynamic semantics

This section serves as supplemental material for Section 3.3.4.

B.1.1 Final judgment

The d final judgment, which describes the set of irreducible expressions in Hazelnut Live internal language, is reproduced in Figure B.1.

B.1.2 Substitution-based evaluation judgment

The evaluation judgment from Hazelnut Live is reproduced in big-step notation in Figure B.2.

B.2 Hazelnut Live substitution-based FAR

This section serves as supplemental material for Section 6.7. The Hazelnut Live formulation of FAR is reproduced in Figure B.3.

$\boxed{d \text{ final}}$	d is final
$\frac{d \text{ boxedval}}{d \text{ final}}$	FBoxedVal
$\frac{d \text{ boxedval}}{d \text{ final}}$	FIndet
$\boxed{d \text{ val}}$	d is a value
$\overline{c \text{ val}}$	VConst
$\overline{\lambda x : \tau. d \text{ val}}$	VLam
$\boxed{d \text{ boxedval}}$	d is a boxed value
$\frac{d \text{ val}}{d \text{ boxedval}}$	BVVal
$\frac{\tau_1 \rightarrow \tau_2 \neq \tau_3 \rightarrow \tau_4}{d(\tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4)}$	$\frac{d \text{ boxedval}}{d(\tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4) \text{ boxedval}}$ BVArrCast
$\frac{d \text{ boxedval} \quad \tau \text{ ground}}{d(\tau \Rightarrow \textcolor{purple}{\langle \rangle}) \text{ boxedval}}$	BVHoleCast
$\boxed{d \text{ indet}}$	d is indeterminate
$\frac{\textcolor{purple}{\langle \rangle}_\sigma^u \text{ indet}}{d \text{ indet}}$	IEHole
$\frac{d_1 \neq d'_1(\tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4) \quad d_1 \text{ indet} \quad d_2 \text{ final}}{d_1 \ d_2 \text{ indet}}$	$\frac{d \text{ final}}{\langle d \rangle_\sigma^u \text{ indet}}$ INEHole
$\frac{d \text{ indet} \quad \tau \text{ ground}}{d(\tau \Rightarrow \textcolor{purple}{\langle \rangle}) \text{ indet}}$	ICastGroundHole
$\frac{d \neq d'(\tau' \Rightarrow \textcolor{purple}{\langle \rangle}) \quad d \text{ indet} \quad \tau \text{ ground}}{d(\textcolor{purple}{\langle \rangle} \Rightarrow \tau) \text{ indet}}$	ICastHoleGround
$\frac{\tau_1 \rightarrow \tau_2 \neq \tau_3 \rightarrow \tau_4 \quad d \text{ indet}}{d(\tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4) \text{ indet}}$	ICastArr
$\frac{d \text{ final} \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{d(\tau_1 \Rightarrow \textcolor{purple}{\langle \rangle} \neq \tau_2) \text{ indet}}$	IFailedCast

Figure B.1: Hazelnut Live final judgment

$\boxed{d \Downarrow d'} d \text{ evaluates to } d'$
$\frac{d \text{ final}}{d \Downarrow d} \text{ EFinal}$
$\frac{d_1 \Downarrow \lambda x : \tau. d'_1 \quad d_2 \Downarrow d'_2 \quad [d'_2/x]d'_1 \Downarrow d}{d_1 \ d_2 \Downarrow d} \text{ EAp}$
$\frac{\tau_1 \rightarrow \tau_2 \neq \tau'_1 \rightarrow \tau'_2 \quad d_1 \Downarrow d'_1(\tau_1 \rightarrow \tau_2 \Rightarrow \tau'_1 \rightarrow \tau'_2) \quad d_2 \Downarrow d'_2 \quad d'_1(d'_2(\tau'_1 \Rightarrow \tau_1))(\tau_2 \Rightarrow \tau'_2) \Downarrow d}{d_1 \ d_2 \Downarrow d} \text{ EApCast}$
$\frac{d \Downarrow d'}{d(\tau \Rightarrow \tau) \Downarrow d'} \text{ ECastId}$
$\frac{d \Downarrow d'}{d(\tau \Rightarrow \textcolor{purple}{\emptyset} \Rightarrow \tau) \Downarrow d'} \text{ ECastSucced}$
$\frac{\tau_1 \neq \tau_2 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad d \Downarrow d'}{d(\tau_1 \Rightarrow \textcolor{purple}{\emptyset} \Rightarrow \tau_2) \Downarrow d(\tau_1 \Rightarrow \textcolor{purple}{\emptyset} \not\Rightarrow \tau_2)} \text{ ECastFail}$
$\frac{\tau \blacktriangleright \tau' \quad d \Downarrow d'}{d(\tau \Rightarrow \textcolor{purple}{\emptyset}) \Downarrow d'(\tau \Rightarrow \tau' \Rightarrow \textcolor{purple}{\emptyset})} \text{ EGround}$
$\frac{\tau \blacktriangleright \tau' \quad d \Downarrow d'}{d(\textcolor{purple}{\emptyset} \Rightarrow \tau) \Downarrow d'(\textcolor{purple}{\emptyset} \Rightarrow \tau' \Rightarrow \tau)} \text{ EExpand}$
$\frac{d \Downarrow d'}{(\textcolor{teal}{d})_\sigma^u \Downarrow (\textcolor{teal}{d}')_\sigma^u} \text{ ENEHole}$

Figure B.2: Hazelnut Live evaluation rules

$\boxed{[\![u/d]\!]d' = d''} \quad d''' \text{ is obtained by filling hole } u \text{ with } d \text{ in } d''$
$\overline{[\![u/d]\!]c = c} \xrightarrow{\text{FillSubConst}} \quad \overline{[\![u/d]\!]x = x} \xrightarrow{\text{FillSubVar}}$
$[\![u/d]\!]\lambda x : \tau. d' = \lambda x : \tau. \overline{[\![u/d]\!]d'} \xrightarrow{\text{FillSubLam}}$
$[\![u/d]\!]d_1 \ d_2 = (\overline{[\![u/d]\!]d_1}) \ (\overline{[\![u/d]\!]d_2}) \xrightarrow{\text{FillSubAp}} \quad \overline{[\![u/d]\!] \emptyset_\sigma^u} = [\![\![u/d]\!] \sigma] d \xrightarrow{\text{FillSubEhole}_1}$
$\overline{[\![u/d]\!] \emptyset_\sigma^d} = \emptyset_\sigma^{u'} \xrightarrow{\text{FillSubEhole}_2} \quad \overline{[\![u/d]\!] \langle d \rangle_\sigma^u} = [\![\![u/d]\!] \sigma] d \xrightarrow{\text{FillSubNEhole}_1}$
$\overline{[\![u/d]\!] \langle d \rangle_\sigma^{u'}} = ([\![u/d]\!] d')_{[\![u/d]\!] \sigma}^{u'} \xrightarrow{\text{FillSubNEhole}_2}$
$[\![u/d]\!]d'(\tau \Rightarrow \tau') = ([\![u/d]\!]d')(\tau \Rightarrow \tau') \xrightarrow{\text{FillSubCast}}$
$[\![u/d]\!]d'(\tau \Rightarrow \emptyset \not\Rightarrow \tau') = ([\![u/d]\!]d')(\tau \Rightarrow \emptyset \not\Rightarrow \tau') \xrightarrow{\text{FillSubFailedCast}}$
$\boxed{[\![u/d]\!] \sigma = \sigma'} \quad \sigma' \text{ is obtained by filling hole } u \text{ with } d \text{ in } \sigma$
$\overline{[\![u/d]\!] \emptyset} = \emptyset \xrightarrow{\text{FillSubEnvNull}} \quad \begin{array}{l} \sigma' = [\![u/d]\!] \sigma \quad d'' = [\![u/d]\!] d' \\ \overline{[\![u/d]\!] \sigma, x \leftarrow d'} = \sigma', x \leftarrow d'' \end{array} \xrightarrow{\text{FillSubEnv}}$

Figure B.3: Hazelnut Live substitution-based FAR

Appendix C

Selected code samples

C.1 Correspondence between theory and code

Table C.1 relates symbols to the corresponding ReasonML module(s) in the implementation. Table C.2 relates algorithms or judgments to the corresponding ReasonML module(s) in the implementation. If the source code for a listed module is present in this Appendix, then the section will be listed as well.

C.2 Relevant code snippets

Relevant code snippets are shown below. Omitted sections are indicated with `/* (...) */`. The source language is ReasonML. Each file represents a module. Files with the `.rei` file

Symbol	Name	Module(s)
e	External expression	<code>UHExp</code>
τ	Type	<code>HTyp</code>
Γ	Typing context	<code>Contexts</code>
d	Internal expression	<code>DHExp</code> (Appendix C.2.1)
σ	(Numbered) environment	<code>EvalEnv</code> , <code>EvalEnvId</code> , <code>VarBstMap</code> (Appendix C.2.2)
H	Hole closure tracking	<code>HoleClosureInfo</code> , <code>HoleClosureInfo_</code> (Appendix C.2.5)
$d_1 \triangleright^* d_2$	Fill diff judgment	<code>DiffDHExp</code> (Appendix C.2.6)

Table C.1: Correspondence between symbols and code

Algorithm	Name	Module(s)
$\sigma \vdash d \Downarrow d'$	Evaluation	Evaluator (Appendix C.2.3)
$d \uparrow d'$	Postprocessing	EvalPostprocess (Appendix C.2.4)
$d_1 \triangleright^v d_2$	Structural diff	DiffDHExp (Appendix C.2.6)
$\llbracket u/d \rrbracket d'$	FAR	FillAndResume (Appendix C.2.6)

Table C.2: Correspondence between algorithms and code

extension are interface files and only contain definitions (the module's public interface) and documentation in the form of comments. Files with the `.re` file extension contain the module's implementation. These code snippets are taken from the `fill-and-resume-backend` branch.

C.2.1 Internal language

```
/* DHExp.rei */
[@deriving sexp]
type t =
  /* Hole types */
  | EmptyHole(MetaVar.t, HoleClosureId.t)
  | NonEmptyHole(ErrStatus.HoleReason.t, MetaVar.t, HoleClosureId.t, t)
  // TODO rename to ExpandingKeyword
  | Keyword(MetaVar.t, HoleClosureId.t, ExpandingKeyword.t)
  | FreeVar(MetaVar.t, HoleClosureId.t, Var.t)
  | InvalidText(MetaVar.t, HoleClosureId.t, string)
  | InconsistentBranches(MetaVar.t, HoleClosureId.t, case)
  /* Generalized closures */
  | Closure(evalenv, bool, t)
  /* Other expressions forms */
  | BoundVar(Var.t)
  | Let(DHPat.t, t, t)
  | Fix(Var.t, HTyp.t, t)
  | Fun(DHPat.t, HTyp.t, t)
  | Ap(t, t)
  | ApBuiltin(string, list(t))
  | BoolLit(bool)
  | IntLit(int)
  | FloatLit(float)
  | BinBoolOp(BinBoolOp.t, t, t)
  | BinIntOp(BinIntOp.t, t, t)
  | BinFloatOp(BinFloatOp.t, t, t)
  | ListNil(HTyp.t)
  | Cons(t, t)
  | Inj(HTyp.t, InjSide.t, t)
  | Pair(t, t)
```

```

    Triv
  ConsistentCase(case)
  Cast(t, HTyp.t, HTyp.t)
  FailedCast(t, HTyp.t, HTyp.t)
  InvalidOpertation(t, InvalidOperationError.t)
and case
| Case(t, list(rule), int)
and rule =
| Rule(DHPat.t, t)
and environment = VarMap.t_(t)
and evalenv = (EvalEnvId.t, VarBstMap.t(result))
and result =
| BoxedValue(t)
| Indet(t);
/* (...) */

/* Used for faster structural equality checking. Structural
   checking may be slow when an expression is large,
   in particular when environments are repeated many times.
   We can optimize checking for structural equality of
   environments simply by checking equality of environment ID's.

Note: assumes that environments with the same EvalEnvId.t
within both expressions are equivalent. This assumption
is true if comparing within a program evaluation (since
EvalEnvId.t numbers don't get reused within a single program
evaluation) or if all the environments are checked to be
equal (see Result.fast.equals).
*/
let fast_equals: (t, t) => bool;

```

Listing 9: DHExp.rei

```

/* DHExp.rei */
/* (...) */

let rec fast_equals = (d1: t, d2: t): bool => {
  switch (d1, d2) {
  /* Primitive forms: regular structural equality */
  | (BoundVar(_), _)
  | (BoolLit(_), _)
  | (IntLit(_), _)
  | (FloatLit(_), _)
  | (ListLit(_), _)
  | (Triv, _) => d1 == d2

  /* Non-hole forms: recurse */
  | (Let(dp1, d11, d21), Let(dp2, d12, d22)) =>
    dp1 == dp2 && fast_equals(d11, d12) && fast_equals(d21, d22)
  }
}

```

```

| (FixF(f1, ty1, d1), FixF(f2, ty2, d2)) =>
  f1 == f2 && ty1 == ty2 && fast_equals(d1, d2)
| (Fun(dp1, ty1, d1), Fun(dp2, ty2, d2)) =>
  dp1 == dp2 && ty1 == ty2 && fast_equals(d1, d2)
| (Ap(d11, d21), Ap(d12, d22))
| (Cons(d11, d21), Cons(d12, d22))
| (Pair(d11, d21), Pair(d12, d22)) =>
  fast_equals(d11, d12) && fast_equals(d21, d22)
| (ApBuiltin(f1, args1), ApBuiltin(f2, args2)) =>
  f1 == f2 && List_for_all2(fast_equals, args1, args2)
| (BinBoolOp(op1, d11, d21), BinBoolOp(op2, d12, d22)) =>
  op1 == op2 && fast_equals(d11, d12) && fast_equals(d21, d22)
| (BinIntOp(op1, d11, d21), BinIntOp(op2, d12, d22)) =>
  op1 == op2 && fast_equals(d11, d12) && fast_equals(d21, d22)
| (BinFloatOp(op1, d11, d21), BinFloatOp(op2, d12, d22)) =>
  op1 == op2 && fast_equals(d11, d12) && fast_equals(d21, d22)
| (Inj(ty1, side1, d1), Inj(ty2, side2, d2)) =>
  ty1 == ty2 && side1 == side2 && fast_equals(d1, d2)
| (Cast(d1, ty11, ty21), Cast(d2, ty12, ty22)) =>
  fast_equals(d1, d2) && ty11 == ty12 && ty21 == ty22
| (FailedCast(d1, ty11, ty21), FailedCast(d2, ty12, ty22)) =>
  fast_equals(d1, d2) && reason1 == reason2
| (InvalidOperation(d1, reason1), InvalidOperation(d2, reason2)) =>
  fast_equals(d1, d2) && reason1 == reason2
| (ConsistentCase(case1), ConsistentCase(case2)) =>
  fast_equals_case(case1, case2)
/* We can group these all into a `|= false` clause; separating
these so that we get exhaustiveness checking. */
| (Let(_, _))
| (FixF(_, _))
| (Fun(_, _))
| (Ap(_, _))
| (ApBuiltin(_, _))
| (Cons(_, _))
| (Pair(_, _))
| (BinBoolOp(_, _))
| (BinIntOp(_, _))
| (BinFloatOp(_, _))
| (Inj(_, _))
| (Cast(_, _))
| (FailedCast(_, _))
| (InvalidOperation(_, _))
| (ConsistentCase(_, _)) => false

/* Hole forms: when checking environments, only check that
environment ID's are equal, don't check structural equality.

(This resolves a performance issue with many nested holes.) */
| (EmptyHole(u1, i1), EmptyHole(u2, i2)) => u1 == u2 && i1 == i2
| (NonEmptyHole(reason1, u1, i1, d1), NonEmptyHole(reason2, u2, i2, d2)) =>
  reason1 == reason2 && u1 == u2 && i1 == i2 && fast_equals(d1, d2)
| (Keyword(u1, i1, kw1), Keyword(u2, i2, kw2)) =>
  u1 == u2 && i1 == i2 && kw1 == kw2
| (FreeVar(u1, i1, x1), FreeVar(u2, i2, x2)) =>
  u1 == u2 && i1 == i2 && x1 == x2

```

```

| (InvalidText(u1, i1, text1), InvalidText(u2, i2, text2)) =>
|   u1 == u2 && i1 == i2 && text1 == text2
| (Closure(ei1, _, _, d1), Closure(ei2, _, _, d2)) =>
/* Cannot use EvalEnv.equals here because it will create a dependency loop. */
|   ei1 == ei2 && fast_equals(d1, d2)
| (InconsistentBranches(u1, i1, case1),
|  InconsistentBranches(u2, i2, case2),
| ) =>
|   u1 == u2 && i1 == i2 && fast_equals_case(case1, case2)
| (EmptyHole(_, _),
|  (NonEmptyHole(_, _),
|   (Keyword(_, _),
|    (FreeVar(_, _),
|     (InvalidText(_, _),
|      (Closure(_, _),
|       (InconsistentBranches(_, _)) => false
|     );
|   );
| and fast_equals_case = (Case(d1, rules1, i1), Case(d2, rules2, i2)) => {
|   fast_equals(d1, d2)
|   && List.length(rules1) == List.length(rules2)
|   && List.for_all2(
|     (Rule(dp1, d1), Rule(dp2, d2)) => dp1 == dp2 && fast_equals(d1, d2),
|     rules1,
|     rules2,
|   )
|   && i1 == i2;
|);

```

Listing 10: DHExp.re

C.2.2 Numbered environments

```

/* VarBstMap.re */
open Sexplib.Std;
module Sexp = Sexplib.Sexp;
include Map.Make(Var);

/* See IntMap */
[@deriving sexp]
type binding('v) = (Var.t, 'v);

let sexp_of_t = (sexp_of_v: 'v => Sexp.t, map: t('v)): Sexp.t =>
  map |> bindings |> sexp_of_list(sexp_of_binding(sexp_of_v));
let t_of_sexp = (v_of_sexp: Sexp.t => 'v, sexp: Sexp.t): t('v) =>
  sexp |> list_of_sexp(binding_of_sexp(v_of_sexp)) |> List.to_seq |> of_seq;

```

Listing 11: VarBstMap.re

```

/* EvalEnv.rei */

/* EvalEnv is an environment (mapping variables to expressions) that
   are used during evaluation. It is different from Environment.t in two ways:
1. It maps Var.t to Evaluator.result (rather than to DHEmp.t)
2. Each EvalEnv has an ID associated with it (if evaluation reaches it),
   or is unreachable. (i.e., a placeholder environment)

Environment.t may be useful in certain cases, namely pattern matching,
when an evaluated result is not needed. EvalEnv is used for environments
during evaluation, including in closures. EvalEnvs are numbered
so that operations on them (e.g., during hole numbering) can be memoized;
the id allows for quick equality checking and allows environments to be
comparable (e.g., so that they can be stored in a map).

Both EvalEnv.t and Environment.t are often named sigma (usually for hole
environments) or env.

This mimicks the VarMap interface on the extended EvalEnv.t type. Most
operations require an EvalState.t parameter, which is used to generate
unique ID's for each environment, and is created using EvalEnv.empty
(at the beginning of evaluation).
*/
[@deriving sexp]
type t = DHExp.evalenv
and result_map = VarBstMap.t(EvaluatorResult.t);

/* Special environment to begin evaluation at the top level
   (empty environment). */
let empty: t;

let id_of_evalenv: t => EvalEnvId.t;
let result_map_of_evalenv: t => result_map;
let environment_of_evalenv: t => Environment.t;
let alist_of_evalenv: t => list((Var.t, EvaluatorResult.t));

let is_empty: t => bool;
let length: t => int;
let to_list: t => list((Var.t, EvaluatorResult.t));
let lookup: (t, Var.t) => option(EvaluatorResult.t);
let contains: (t, Var.t) => bool;

/* Equals only needs to check environment ID's.
   (faster than structural equality checking.) */
let equals: (t, t) => bool;

```

```

/* these functions require an `EvalState.t` because they generate a new
`EvalEnvId.t` */
let extend: (EvalState.t, t, (Var.t, EvaluatorResult.t)) => (EvalState.t, t);
let map:
  (EvalState.t, (Var.t, EvaluatorResult.t) => EvaluatorResult.t, t) =>
  (EvalState.t, t);
let filter:
  (EvalState.t, (Var.t, EvaluatorResult.t) => bool, t) => (EvalState.t, t);
/* union(new_env, env) extends env with new_env (same argument order
as in VarMap.union) */
let union: (EvalState.t, t, t) => (EvalState.t, t);
/* same as map, but doesn't assign a new ID. (This is used when
transforming an environment, such as in the closure->lambda stage
after evaluation. More functions may be added like this as-needed
for similar purposes.) */
let map_keep_id: ((Var.t, EvaluatorResult.t) => EvaluatorResult.t, t) => t;

```

Listing l2: EvalEnv.rei

```

/* EvalEnv.rei */

[@deriving sexp]
type t = DHExp.evalenv;

[@deriving sexp]
type result_map = VarBstMap.t(EvaluatorResult.t);

/* Environment with a special `EvalEnvId.t` of zero. */
let empty: t = (EvalEnvId.empty, VarBstMap.empty);

let id_of_evalenv = ((ei, _): t): EvalEnvId.t => ei;

let environment_of_evalenv = ((_, result_map): t): Environment.t =>
  result_map
  [> VarBstMap.bindings
   > List.map(((x, res: EvaluatorResult.t)) =>
      switch (res) {
        | Indet(d)
        | BoxedValue(d) => (x, d)
      }
    );
  ];

let result_map_of_evalenv = ((_, result_map): t): result_map => result_map;

let alist_of_evalenv =
  ((_, result_map): t): list((Var.t, EvaluatorResult.t)) =>
  result_map [> VarBstMap.bindings];

let is_empty = (env: t) => VarBstMap.is_empty(result_map_of_evalenv(env));

```

```

let equals = (env1: t, env2: t): bool =>
  id_of_evalenv(env1) == id_of_evalenv(env2);

let extend =
  (es: EvalState.t, env: t, (x, a): (Var.t, EvaluatorResult.t)) =>
  (EvalState.t, t) => {
    let (es, ei) = es |> EvalState.next_evalenvid;
    (es, (ei, VarBstMap.add(x, a, result_map_of_evalenv(env))))};
};

let union = (es: EvalState.t, env1: t, env2: t): (EvalState.t, t) => {
  let (es, ei) = es |> EvalState.next_evalenvid;
  {
    es,
    {
      ei,
      VarBstMap.union(
        (., dr, _) => Some(dr),
        result_map_of_evalenv(env1),
        result_map_of_evalenv(env2),
        ),
      );
    };
};

let lookup = (env: t, x) =>
  env |> result_map_of_evalenv |> VarBstMap.find_opt(x);

let contains = (env: t, x) =>
  env |> result_map_of_evalenv |> VarBstMap.mem(x);

let map = (es: EvalState.t, f, env: t): (EvalState.t, t) => {
  let (es, ei) = es |> EvalState.next_evalenvid;
  (es, (ei, VarBstMap.mapi(f, result_map_of_evalenv(env))));

let map_keep_id = (f, env: t): t => (
  id_of_evalenv(env),
  VarBstMap.mapi(f, result_map_of_evalenv(env)),
);

let filter = (es: EvalState.t, f, env: t): (EvalState.t, t) => {
  let (es, ei) = es |> EvalState.next_evalenvid;
  (es, (ei, VarBstMap.filter(f, result_map_of_evalenv(env))));

let length = (env: t): int =>
  VarBstMap.cardinal(result_map_of_evalenv(env));

let to_list = (env: t): list((Var.t, EvaluatorResult.t)) =>
  env |> result_map_of_evalenv |> VarBstMap.bindings;
}

```

Listing 13: EvalEnv.re

```
/* EvalEnvId.rei */
/* Identifier for an EvalEnv.t, generated by EvalEnvIdGen */
[Deriving sexp]
type t = int;
/* A special value used only by 'EvalEnv.empty'. */
let empty: t;
```

Listing 14: EvalEnvId.rei

```
/* EvalEnvId.re */
open Sexplib.Std;
[Deriving sexp]
type t = int;
let empty: t = 0;
```

Listing 15: EvalEnvId.re

C.2.3 Evaluation

```
/* Evaluator.rei */
/* (...) */
let evaluate:
  (EvalState.t, EvalEnv.t, DHExp.t) => (EvalState.t, EvaluatorResult.t);
/* (...) */
```

Listing 16: Evaluator.rei

```
/* Evaluator.re */
/* (...) */
```

```

let rec evaluate =
  (es: EvalState.t, env: EvalEnv.t, d: DExp.t)
  : (EvalState.t, EvaluatorResult.t) => (
    /* Update evaluation statistics */
    let es' = es >> EvalState.inc_steps;
    switch (d) {
      | BoundVar(x) =>
        let dr =
          x
          |> EvalEnv.lookup(env)
          |> OptUtil.get_... =>
            raise(EvaluatorError.Exception(FreeInvalidVar(x)))
        );
      switch (dr) {
        | BoredValue(FixF(..) as d) => evaluate(es, env, d)
        | _ => (es, dr)
      };
      | Let(dp, d1, d2) =>
        switch (evaluate(es, env, d1)) {
          | (es, BoxedValue(d1))
          | (es, Indet(d1)) =>
            switch (matches(dp, d1)) {
              | Indet
              | DoesNotMatch => (es, Indet(Closure(env, false, Let(dp, d1, d2))))
              | Matches(env') =>
                let (es, env) = extend_evalenv_with_env(es, env', env);
                evaluate(es, env, d2);
            }
        }
        | FixF(f, ty, d) =>
          switch (evaluate(es, env, d)) {
            | (es, BoxedValue(Closure(env', false, Fun(..) as d'') as d'')) =>
              let (es, env') =
                EvalEnv.extend(es, env', (f, BoxedValue(FixF(.., ty, d''))));
              (es, BoxedValue(Closure(env'', false, d'')));
            | _ => raise(EvaluatorError.Exception(EvaluatorError.FixFWithoutLambda))
          }
          Fun(..) => (es, BoxedValue(Closure(env, false, d)))
        Ap(d1, d2) =>
        switch (evaluate(es, env, d1)) {
          | (es, BoxedValue(Closure(closure_env, false, Fun(dp, .., d3)) as d1)) =>
            switch (evaluate(es, env, d2)) {
              | (es, BoxedValue(d2))
              | (es, Indet(d2)) =>
                switch (matches(dp, d2)) {
                  | DoesNotMatch
                  | Indet => (es, Indet(Ap(d1, d2)))
                  | Matches(env') =>
                    // evaluate a closure: extend the closure environment with the
                    // new bindings introduced by the function application.
                    let (es, env) = extend_evalenv_with_env(es, env', closure_env);
                    evaluate(es, env, d3);
                }
            }
        }
      }
    }
  )

```

```

}
| (es, BoxedValue(Cast(d1', Arrow(ty1, ty2), Arrow(ty1', ty2')))) =>
| (es, Indet(Cast(d1', Arrow(ty1, ty2), Arrow(ty1', ty2')))) =>
switch (evaluate(es, env, d2)) {
| (es, BoxedValue(d2')) =>
| (es, Indet(d2')) =>
/* ap cast rule */
evaluate(es, env, Cast(Ap(d1', Cast(d2', ty1', ty1)), ty2, ty2'))
}
| (., BoxedValue(d1')) =>
raise(EvaluatorError.Exception(InvalidBoxedFun(d1')))
| (es, Indet(d1')) =>
switch (evaluate(es, env, d2)) {
| (es, BoxedValue(d2')) =>
| (es, Indet(d2')) =>> (es, Indet(Ap(d1', d2'))))
}
}
ApBuiltin(ident, args) => evaluate_ap_builtin(es, env, ident, args)
ListNil(_)
BoolLit(_)
IntLit(_)
FloatLit(_)
Triv => (es, BoxedValue(d))
BinBoolOp(op, d1, d2) =>
switch (evaluate(es, env, d1)) {
| (es, BoxedValue(BoolLit(b1) as d1')) =>
switch (eval_val_bin_bool_op_short_circuit(op, b1)) {
| Some(b3) => (es, b3)
| None =>
switch (evaluate(es, env, d2)) {
| (es, BoxedValue(BoolLit(b2))) => (
es,
BoxedValue(eval_val_bin_bool_op(op, b1, b2)),
)
| (., BoxedValue(d2')) =>
raise(EvaluatorError.Exception(InvalidBoxedBoolLit(d2'))))
| (es, Indet(d2')) =>> (es, Indet(BinBoolOp(op, d1', d2'))))
}
}
| (., BoxedValue(d1')) =>
raise(EvaluatorError.Exception(InvalidBoxedBoolLit(d1'))))
| (es, Indet(d1')) =>
switch (evaluate(es, env, d2)) {
| (es, BoxedValue(d2')) =>
| (es, Indet(d2')) =>> (es, Indet(BinBoolOp(op, d1', d2'))))
}
}
BinIntOp(op, d1, d2) =>
switch (evaluate(es, env, d1)) {
| (es, BoxedValue(IntLit(n1) as d1')) =>
switch (evaluate(es, env, d2)) {
| (es, BoxedValue(IntLit(n2))) =>
switch (op, n1, n2) {
| (Divide, _, 0) => (

```

```

    es,
    Indet(
      InvalidOperation(
        BinIntOp(op, IntLit(n1), IntLit(n2)),
        DivideByZero,
        ),
      ),
    ) => (es, BoxedValue(eval_bin_int_op(op, n1, n2)))
  }
| (., BoxedValue(d2')) =>
  raise(EvaluatorError.Exception(InvalidBoxedIntLit(d2')))
| (es, Indet(d2')) => (es, Indet(BinIntOp(op, d1', d2')))

| (., BoxedValue(d1')) =>
  raise(EvaluatorError.Exception(InvalidBoxedIntLit(d1')))

| (es, Indet(d1')) =>
  switch (evaluate(es, env, d2)) {
  | (es, BoxedValue(d2')) =>
    | (es, Indet(d2')) => (es, Indet(BinIntOp(op, d1', d2')))

  }
| BinFloatOp(op, d1, d2) =>
  switch (evaluate(es, env, d1)) {
  | (es, BoxedValue(FloatLit(f1) as d1')) =>
    switch (evaluate(es, env, d2)) {
    | (es, BoxedValue(FloatLit(f2))) => (
      es,
      BoxedValue(eval_bin_float_op(op, f1, f2)),
    )
    | (., BoxedValue(d2')) =>
      raise(EvaluatorError.Exception(InvalidBoxedFloatLit(d2')))

    | (es, Indet(d2')) => (es, Indet(BinFloatOp(op, d1', d2')))

    }
  | (., BoxedValue(d1')) =>
    raise(EvaluatorError.Exception(InvalidBoxedFloatLit(d1')))

  | (es, Indet(d1')) =>
    switch (evaluate(es, env, d2)) {
    | (es, BoxedValue(d2')) =>
      | (es, Indet(d2')) => (es, Indet(BinFloatOp(op, d1', d2')))

    }
  }

| Inj(ty, side, d1) =>
  switch (evaluate(es, env, d1)) {
  | (es, BoxedValue(d1')) => (es, BoxedValue(Inj(ty, side, d1')))

  | (es, Indet(d1')) => (es, Indet(Inj(ty, side, d1')))

  }

Pair(d1, d2) =>
let (es, d1') = evaluate(es, env, d1);
let (es, d2') = evaluate(es, env, d2);
switch (d1', d2') {
| (Indet(d1), Indet(d2))
| (Indet(d1), BoxedValue(d2))
| (BoxedValue(d1), Indet(d2)) => (es, Indet(Pair(d1, d2)))
}

```

```

| (BoxedValue(d1), BoxedValue(d2)) => (es, BoxedValue(Pair(d1, d2)))
},
Cons(d1, d2) =>
let (es, d1') = evaluate(es, env, d1);
let (es, d2') = evaluate(es, env, d2);
switch (d1', d2') {
| (Indet(d1), Indet(d2))
| (Indet(d1), BoxedValue(d2))
| (BoxedValue(d1), Indet(d2)) => (es, Indet(Cons(d1, d2)))
| (BoxedValue(d1), BoxedValue(d2)) => (es, BoxedValue(Cons(d1, d2)))
};
ConsistentCase(Case(d1, rules, n)) =>
evaluate_case(es, env, None, d1, rules, n)

/* Generalized closures evaluate to themselves. Only
lambda closures are BoxedValues; other closures are all Indet. */
| Closure(_, false, d') =>
switch (d') {
| Fun(_) => (es, BoxedValue(d))
| _ => (es, Indet(d))
}

/* For purposes of fill-and-resume, 'Closure' expressions may not be final.
All closures are marked with a 're_eval' flag (the second parameter)
before resuming evaluation during fill-and-resume. After a Closure is
evaluated for the first time, then it does not need to be re-evaluated
when encountered in the future.

In addition, if a closure marks a filled hole, then the evaluation
may be memoized by hole closure. (See TODO, below.) */

TODO: memoize the filling of the same hole instance. To do this, store
the hole instance in the 're_eval' field (make 're_eval' not only a
boolean flag) and store a mapping from ids to results in the
'EvalState.t' (replacing 'FARInfo.t').

*/
| Closure(env', true, d') => evaluate(es, env', d')

/* Hole expressions. Wrap in closure. */
| EmptyHole(_)
| FreeVar(_)
| Keyword(_)
| InvalidText(_) => (es, Indet(Closure(env, false, d)))
| InconsistentBranches(u, i, Case(d1, rules, n)) =>
evaluate_case(es, env, Some((u, i)), d1, rules, n)
| NonEmptyHole(reason, u, i, di) =>
switch (evaluate(es, env, di)) {
| (es, BoxedValue(di'))
| (es, Indet(di')) => (
es,
Indet(Closure(env, false, NonEmptyHole(reason, u, i, di'))),
)
}
}

```

```

/* Cast calculus */
| Cast(d1, ty, ty') =>
switch (evaluate(es, env, d1)) {
| es_BoxedValue(d1') as result) =>
switch (ground_cases_of(ty), ground_cases_of(ty')) {
| (Hole, Hole) => (es, result)
| (Ground, Ground) =>
/* if two types are ground and consistent, then they are eq */
(es, result)
| (Ground, Hole) =>
/* can't remove the cast or do anything else here, so we're done */
(es, BoxedValue(Cast(d1', ty, ty')))

| (Hole, Ground) =>
/* by canonical forms, d1' must be of the form d<ty' -> ? */
switch (d1') {
| Cast(d1'', ty'', Hole) =>
if (HTyp.eq(ty'', ty')) {
(es, BoxedValue(d1''));
} else {
(es, Indet(FailedCast(d1', ty, ty')));
}
| _ =>
raise(EvaluatorError.Exception(CastBVHoleGround(d1')));
}
| (Hole, NotGroundOrHole(ty'_grounded)) =>
/* ITExpand rule */
let d' = DHExp.Cast(Cast(d1', ty, ty'_grounded), ty'_grounded, ty');
evaluate(es, env, d');
| (NotGroundOrHole(_), Hole) =>
/* ITGround rule */
let d' = DHExp.Cast(Cast(d1', ty, ty'_grounded), ty'_grounded, ty');
evaluate(es, env, d');
| (Ground, NotGroundOrHole(_))
| (NotGroundOrHole(_), Ground) =>
/* can't do anything when casting between diseq, non-hole types */
(es, BoxedValue(Cast(d1', ty, ty')))

| (NotGroundOrHole(_), NotGroundOrHole(_)) =>
/* they might be eq in this case, so remove cast if so */
if (HTyp.eq(ty, ty')) {
(es, result);
} else {
(es, BoxedValue(Cast(d1', ty, ty')));
}
}

| (es, Indet(d1') as result) =>
switch (ground_cases_of(ty), ground_cases_of(ty')) {
| (Hole, Hole) => (es, result)
| (Ground, Ground) =>
/* if two types are ground and consistent, then they are eq */
(es, result)
| (Ground, Hole) =>
/* can't remove the cast or do anything else here, so we're done */
(es, Indet(Cast(d1', ty, ty')))

| (Hole, Ground) =>

```

```

switch (d1) {
| Cast(d1'', ty'', Hole) =>
  if (HTyp.eq(ty'', ty'')) {
    (es, Indet(d1''));
  } else {
    (es, IndetFailedCast(d1'', ty, ty'')));
  }
| _ => (es, Indet(Cast(d1', ty, ty'))))

| (Hole, NotGroundOrHole(ty'_grounded)) =>
/* IIEexpand rule */
let d' = DHExp.Cast(Cast(d1', ty, ty'_grounded), ty'_grounded, ty');
evaluate(es, env, d');

| (NotGroundOrHole(ty_grounded), Hole) =>
/* ITIGround rule */
let d' = DHExp.Cast(Cast(d1', ty, ty_groundered), ty_groundered, ty');
evaluate(es, env, d');

| (Ground, NotGroundOrHole(_))
| (NotGroundOrHole(_), Ground) =>
/* can't do anything when casting between diseq, non-hole types */
(es, Indet(Cast(d1', ty, ty')))

| (NotGroundOrHole(_), NotGroundOrHole(_)) =>
/* it might be eq in this case, so remove cast if so */
if (HTyp.eq(ty, ty'')) {
  (es, result);
} else {
  (es, Indet(Cast(d1', ty, ty')));
}
}

| FailedCast(d1, ty, ty') =>
switch (evaluate(es, env, d1)) {
| (es, BoxedValue(d1''))
| (es, Indet(d1'')) => (es, IndetFailedCast(d1', ty, ty''))
}
| InvalidOperation(d, err) => (es, Indet(InvalidOperation(d, err)))
};

and evaluate_case =
(
  es: EvalState.t,
  env: EvalEnv.t,
  inconsistent_info: option(HoleClosure.t),
  scrut: DHExp.t,
  rules: list(DHExp.rule),
  current_rule_index: int,
)
: (EvalState.t, EvaluatorResult.t) =>
switch (evaluate(es, env, scrut)) {
| (es, BoxedValue(scrut))
| (es, Indet(scrut)) =>
  switch (List.nth_opt(rules, current_rule_index)) {
  | None =>
    let case = DHExp.Case(scrut, rules, current_rule_index);
  }
}

```

```

(
  es,
  switch (inconsistent_info) {
    | None => Indet(Closure(env, false, ConsistentCase(case)))
    | Some((u, i)) =>
      Indet(Closure(env, false, InconsistentBranches(u, i, case)))
  },
),
| Some(Rule(dp, d)) =>
  switch (matches(dp, scrut)) {
    | Indet =>
      let case = DHExp.Case(scrut, rules, current_rule_index);
      (
        es,
        switch (inconsistent_info) {
          | None => Indet(Closure(env, false, ConsistentCase(case)))
          | Some((u, i)) =>
            Indet(Closure(env, false, InconsistentBranches(u, i, case)))
        },
      );
    | Matches(env') =>
      // extend environment with new bindings introduced
      let (es, env) = extend_evalenv_with_env(es, env', env);
      evaluate(es, env, d);
      // by the rule and evaluate the expression.
    | DoesNotMatch =>
      evaluate_case(
        es,
        env,
        inconsistent_info,
        scrut,
        rules,
        current_rule_index + 1,
      )
  }
}

/* This function extends an EvalEnv.t with new bindings
   (an Environment.t from match()). We need to wrap the new bindings
   in a final judgment (BoxedValue or Indet), so we call evaluate()
   on it again, but it shouldn't change the value of the expression. */
and extend_evalenv_with_env =
  (es : EvalState.t, new_bindings: Environment.t, to_extend: EvalEnv.t)
  : (EvalState.t, EvalEnv.t) => {
  let (es, ei) = es |> EvalState.next_evalenvid;
  let result_map =
    List.fold_left(
      (new_env, (x, d)) => {
        /* The value of environment doesn't matter here */
        let (_, dr) = evaluate(es, EvalEnv.empty, d);
        VarBstMap.add(x, dr, new_env);
      },
      EvalEnv.result_map_of_evalenv(to_extend),
    )
}

```

```

        new_bindings,
    );
(es, (ei, result_map));
}

/* Evaluate the application of a built-in function. */
and evaluate_ap_builtin =
(es: EvalState.t, env: EvalEnv.t, ident: string, args: list(DHExp.t))
: (EvalState.t, EvaluatorResult.t) => {
  switch (Builtins.lookup_form(ident)) {
    Some((eval_...)) => eval(es, env, args, evaluate)
    | None => raise(EvaluatorError.Exception(InvalidBuiltIn(ident)))
  };
};
}

```

Listing 17: Evaluator.re

C.2.4 Postprocessing

```

/* EvalPostprocess.rei */

/* Postprocesses the evaluation result. This has two functions:
   - Match the evaluation result generated by evaluation with substitution.
     This means to continue evaluation within expressions for which evaluation
     has not reached (e.g., lambda expression bodies, unmatched case and let
     expression bodies), by looking up bound variables and assigning hole
     environments.
   - Number holes and generate a HoleClosureInfo.t that holds information
     about all unique hole closures in the result.

The postprocessing steps are partially memoized by environments. (Only
memorized among hole instances which share the same environment.)

Algorithmically, this algorithm begins in the evaluated region of the
evaluation result inside the "evaluation boundary" (pp_eval),
and continues to the region outside the evaluation boundary (pp_uneval).
*/
let postprocess: DHExp.t => (HoleClosureInfo.t, DHExp.t);

```

Listing 18: EvalPostprocess.rei

```

/* EvalPostprocess.rei */

/* Postprocess outside evaluation boundary */
let rec pp_uneval =

```

```

    <
      hci: HoleClosureInfo_.t,
      env: EvalEnv.t,
      d: DHExp.t,
      parent: HoleClosureParents.t_,
    >
    : (HoleClosureInfo_.t, DHExp.t) => {
  switch (d) {
    /* Bound variables should be looked up within the closure
     environment. If lookup fails, then variable is not bound. */
    | BoundVar(x) =>
      switch (EvalEnv.lookup(env, x)) {
        | Some(Indet(d')) =>
          let (hci, d'') = pp_eval(hci, d', parent);
          (hci, d'');
        | None => (hci, d)
      }
    /* Non-hole expressions: expand recursively */
    | BoolLit(_) |
    | IntLit(_) |
    | FloatLit(_) |
    | ListNil(_) |
    | Triv => (hci, d)
    | Let(dp, d1, d2) =>
      let (hci, d1') = pp_uneval(hci, env, d1, parent);
      let (hci, d2') = pp_uneval(hci, env, d2, parent);
      (hci, Let(dp, d1', d2'));
    | FixF(f, ty, d1) =>
      let (hci, d1') = pp_uneval(hci, env, d1, parent);
      (hci, FixF(f, ty, d1'));
    | Fun(dp, ty, d1') =>
      let (hci, d1') = pp_uneval(hci, env, d1, parent);
      (hci, Fun(dp, ty, d1'));
    | Ap(d1, d2) =>
      let (hci, d1') = pp_uneval(hci, env, d1, parent);
      let (hci, d2') = pp_uneval(hci, env, d2, parent);
      (hci, Ap(d1', d2'));
    | ApBuiltIn(f, args) =>
      let (hci, args') =
        List.fold_right(
          (arg, (hci, args)) => {
            let (hci, arg') = pp_uneval(hci, env, arg, parent);
            (hci, [arg', ...args]);
          },
          args,
          (hci, []);
        );
      (hci, ApBuiltIn(f, args'));
    | BinBoolOp(op, d1, d2) =>
      let (hci, d1') = pp_uneval(hci, env, d1, parent);
      let (hci, d2') = pp_uneval(hci, env, d2, parent);
      (hci, BinBoolOp(op, d1', d2'));
  }
}

```

```

| BinIntOp(op, d1, d2) =>
let (hci, d1') = pp_uneval(hci, env, d1, parent);
let (hci, d2') = pp_uneval(hci, env, d2, parent);
(hci, BinIntOp(op, d1', d2'));
| BinIntOp(op, d1, d2) =>
let (hci, d1') = pp_uneval(hci, env, d1, parent);
let (hci, d2') = pp_uneval(hci, env, d2, parent);
(hci, BinIntOp(op, d1', d2'));
| BinFloatOp(op, d1, d2) =>
let (hci, d1') = pp_uneval(hci, env, d1, parent);
let (hci, d2') = pp_uneval(hci, env, d2, parent);
(hci, BinFloatOp(op, d1', d2'));
| Cons(d1, d2) =>
let (hci, d1') = pp_uneval(hci, env, d1, parent);
let (hci, d2') = pp_uneval(hci, env, d2, parent);
(hci, Cons(d1', d2'));
| Inj(ty, side, d') =>
let (hci, d') = pp_uneval(hci, env, d', parent);
(hci, Inj(ty, side, d'));
| Pair(d1, d2) =>
let (hci, d1') = pp_uneval(hci, env, d1, parent);
let (hci, d2') = pp_uneval(hci, env, d2, parent);
(hci, Pair(d1', d2'));
| Cast(d', ty1, ty2) =>
let (hci, d') = pp_uneval(hci, env, d', parent);
(hci, Cast(d', ty1, ty2));
| FailedCast(d', ty1, ty2) =>
let (hci, d') = pp_uneval(hci, env, d', parent);
(hci, FailedCast(d', ty1, ty2));
| InvalidOperation(d', reason) =>
let (hci, d') = pp_uneval(hci, env, d', parent);
(hci, InvalidOperation(d', reason));
| ConsistentCase(case scrut, rules, i) =>
let (hci, scrut') = pp_uneval(hci, env, scrut, parent);
let (hci, rules') = pp_uneval_rules(hci, env, rules, parent);
(hci, ConsistentCase(case scrut', rules', i));

/* Closures shouldn't exist inside other closures */
| Closure(_) => raise(EvalPostprocessError.Exception(ClosureInsideClosure))

/* Hole expressions:
   - Use the closure environment as the hole environment.
   - Number the hole closure appropriately.
   - Recurse through inner expression (if any).

Note: we still have to recurse through the hole boundary in order
to set the correct hole parents; however, this is only at most
one depth of repeated traversal through the environment
*/
| EmptyHole(u, _) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env) => (
  hci,
  Closure(env, false, EmptyHole(u, i)),
)
| NewClosure(hci, i) =>
  let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
}

```

```

let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
  (hci, Closure(env, false, EmptyHole(u, i)));
};

NonEmptyHole(reason, u, _, d') =>
let (hci, d'') = pp_uneval(hci, env, d', parent);
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env) => (
  hci,
  Closure(env, false, NonEmptyHole(reason, u, i, d'')));
)
| NewClosure(hci, i) =>
let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
(hci, Closure(env, false, NonEmptyHole(reason, u, i, d'')));
};

Keyword(u, _, kw) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env) => (
  hci,
  Closure(env, false, Keyword(u, i, kw)));
)
| NewClosure(hci, i) =>
let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
(hci, Closure(env, false, Keyword(u, i, kw)));
};

FreeVar(u, _, x) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env) => (
  hci,
  Closure(env, false, FreeVar(u, i, x)));
)
| NewClosure(hci, i) =>
let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
(hci, Closure(env, false, FreeVar(u, i, x)));
};

InvalidText(u, _, text) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env) => (
  hci,
  Closure(env, false, InvalidText(u, i, text)));
)
| NewClosure(hci, i) =>
let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
(hci, Closure(env, false, InvalidText(u, i, text)));
};

InconsistentBranches(u, _, Case(d', rules, i)) =>
let (hci, d'') = pp_uneval(hci, env, d', parent);

```

```

let (hci, rules') = pp_uneval_rules(hci, env, rules, parent);
let case' = DHExp.Case(d'', rules', i);
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env, parent);
switch `hc_id_res` {
| ExistingClosure(hci, i, env) => (
  hci,
  Closure(env, false, InconsistentBranches(u, i, case'))),
|
| NewClosure(hci, i) =>
  let (hci, env) = pp_eval_hole_env(hci, env, (u, i));
  let hci = HoleClosureInfo_.update_hc_env(hci, u, env);
  (hci, Closure(env, false, InconsistentBranches(u, i, case'))));
},
};

and pp_uneval_rules =
(
  hci: HoleClosureInfo_.t,
  env: EvalEnv.t,
  rules: list(DHExp.rule),
  parent: HoleClosureParents.t.,
)
: (HoleClosureInfo_.t, list(DHExp.rule)) =>
List.fold_right(
  (DHExp.Rule(dp, d), (hci, rules)) => {
    let (hci, d') = pp_uneval(hci, env, d, parent);
    (hci, [DHExp.Rule(dp, d'), ...rules]);
  },
  rules,
  (hci, []),
)

/* Postprocess inside evaluation boundary */
and pp_eval =
(hci: HoleClosureInfo_.t, d: DHExp.t, parent: HoleClosureParents.t.)
: (HoleClosureInfo_.t, DHExp.t) =>
switch `(d)` {
/* Non-hole expressions: recurse through subexpressions */
| BoolLit(_)
| IntLit(_)
| FloatLit(_)
| ListNil(_)
| Triv => (hci, d)
| FixF(f, ty, d1) =>
  let (hci, d1') = pp_eval(hci, d1, parent);
  (hci, FixF(f, ty, d1'));
| Ap(d1, d2) =>
  let (hci, d1') = pp_eval(hci, d1, parent);
  let (hci, d2') = pp_eval(hci, d2, parent);
  (hci, Ap(d1', d2')));
| App_builtin(f, args) =>
  let (hci, args') =
    List.fold_right(

```

```

(arg, (hci, args)) => {
  let (hci, arg') = pp_eval(hci, arg, parent);
  (hci, [arg', ...args]);
}
args,
(hci, □),
);
(hci, ApBuiltin(`f, args'));
BinBoolOp(op, d1, d2) =>
let (hci, d1') = pp_eval(hci, d1, parent);
let (hci, d2') = pp_eval(hci, d2, parent);
(hci, BinBoolOp(op, d1', d2'));
BinIntOp(op, d1, d2) =>
let (hci, d1') = pp_eval(hci, d1, parent);
let (hci, d2') = pp_eval(hci, d2, parent);
(hci, BinIntOp(op, d1', d2'));
BinFloatOp(op, d1, d2) =>
let (hci, d1') = pp_eval(hci, d1, parent);
let (hci, d2') = pp_eval(hci, d2, parent);
(hci, BinFloatOp(op, d1', d2'));
| Cons(d1, d2) =>
let (hci, d1') = pp_eval(hci, d1, parent);
let (hci, d2') = pp_eval(hci, d2, parent);
(hci, Cons(d1', d2'));
| Inj(ty, side, d') =>
let (hci, d') = pp_eval(hci, d', parent);
(hci, Inj(ty, side, d'));
| Pair(d1, d2) =>
let (hci, d1') = pp_eval(hci, d1, parent);
let (hci, d2') = pp_eval(hci, d2, parent);
(hci, Pair(d1', d2'));
| Cast(d', ty1, ty2) =>
let (hci, d') = pp_eval(hci, d', parent);
(hci, Cast(d', ty1, ty2));
| FailedCast(d', ty1, ty2) =>
let (hci, d') = pp_eval(hci, d', parent);
(hci, FailedCast(d', ty1, ty2));
| InvalidOperation(d', reason) =>
let (hci, d') = pp_eval(hci, d', parent);
(hci, InvalidOperation(d', reason));
/* Bound variables should not appear outside holes or closures */
| BoundVar(x) =>
raise(EvalPostprocessError.Exception(BoundVarOutsideClosure(x)));
/* Lambda should not appear outside closure in evaluated result */
| Let(_)
| ConsistentCase(_)
| Fun(_)
| EmptyHole(_)
| NonEmptyHole(_)
| Keyword(_)
| FreeVar(_)
| InvalidText(_)

```

```

| InconsistentBranches(_) =>
raise(EvalPostprocessError.Exception(UnevalOutsideClosure))

/* Closure */
| Closure(env', _, d') =>
switch(d') {
| Fun(dp, ty, d'') =>
let(hci, d'') = pp_uneval(hci, env', d'', parent);
(hci, Fun(dp, ty, d''));
| Let(dp, d1, d2, _) =>
/* d1 should already be evaluated, d2 is not */
let(hci, scrut') = pp_eval(hci, d1, parent);
let(hci, d2') = pp_uneval(hci, env', d2, parent);
(hci, Let(dp, d1', d2'));
| ConsistentCase(Case{scrut, rules, i}) =>
/* scrut should already be evaluated, rule bodies are not */
let(hci, scrut') = pp_eval(hci, scrut, parent);
let(hci, rules') = pp_uneval_rules(hci, env', rules, parent);
(hci, ConsistentCase(Case{scrut', rules', i}));
}

/* Holes: should be left in closures in the result */
| EmptyHole(u, _) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch(hc_id_res) {
| ExistClosure(hci, i, env') => (
  hci,
  Closure(env', false, EmptyHole(u, i)),
)
| NewClosure(hci, i) =>
let(hci, env') = pp_eval_hole_env(hci, env', (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(hci, Closure(env', false, EmptyHole(u, i)));
};

| NonEmptyHole(reason, u, _, d') =>
let(hci, d'') = pp_eval(hci, d', parent);
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch(hc_id_res) {
| ExistClosure(hci, i, env') => (
  hci,
  Closure(env', false, NonEmptyHole(reason, u, i, d'')),
)
| NewClosure(hci, i) =>
let(hci, env') = pp_eval_hole_env(hci, env', (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(hci, Closure(env', false, NonEmptyHole(reason, u, i, d'')));
};

| Keyword(u, _, kw) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch(hc_id_res) {
| ExistClosure(hci, i, env') => (
  hci,
  Closure(env', false, Keyword(u, i, kw)),
)
| NewClosure(hci, i) =>

```

```

let (hci, env') = pp_eval_hole_env(hci, env', (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(hci, Closure(env', false, Keyword(u, i, kw)));
};

| FreeVar(u, _, x) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env') => (
  hci,
  Closure(env', false, FreeVar(u, i, x)),
)
| NewClosure(hci, i) =>
let (hci, env') = pp_eval_hole_env(hci, env', (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(hci, Closure(env', false, FreeVar(u, i, x)));
};
| InvalidText(u, _, text) =>
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env') => (
  hci,
  Closure(env', false, InvalidText(u, i, text)),
)
| NewClosure(hci, i) =>
let (hci, env') = pp_eval_hole_env(hci, env', (u, i));
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(hci, Closure(env', false, InvalidText(u, i, text)));
};
| InconsistentBranches(u, _, Case(d', rules, case_i)) =>
let (hci, d') = pp_eval(hci, d', parent);
let hc_id_res = HoleClosureInfo_.get_hc_id(hci, u, env', parent);
switch (hc_id_res) {
| ExistClosure(hci, i, env') =>
let (hci, rules') = pp_uneval_rules(hci, env', rules, parent);
(
  hci,
  Closure(
    env',
    false,
    InconsistentBranches(u, i, Case(d'', rules', case_i)),
  ),
);
| NewClosure(hci, i) =>
let (hci, env') = pp_eval_hole_env(hci, env', (u, i));
let (hci, rules') = pp_uneval_rules(hci, env', rules, parent);
let hci = HoleClosureInfo_.update_hc_env(hci, u, env');
(
  hci,
  Closure(
    env',
    false,
    InconsistentBranches(u, i, Case(d'', rules', case_i)),
  ),
);
}

```

```

    };

    /* Fill-and-resume: may be possible to have other expression
     * types inside a closure. */
    | _ => pp_uneval(hci, env', d', parent)
  )
  /* Hole expressions:
   * - Fix environment recursively.
   * - Number the hole closure appropriately.
   * - Recurse through subexpressions if applicable. */
}

/* Apply pp_eval to each expression in sigma,
threading hci throughout.

hc is the current HoleClosure t, which will be set as the
parent of any holes directly in the subexpressions
*/
and pp_eval_hole_env =
  (hci: HoleClosureInfo_.t, sigma: EvalEnv.t, parent_hc: HoleClosure.t)
  : (HoleClosureInfo_.t, EvalEnv.t) => {
let ei = sigma >| EvalEnv.id_of_evalenv;
let (hci, result_map) =
  VarBstMap.fold(
    (x, dr: EvaluatorResult.t, (hci, new_env)) => {
      let (hci, dr, EvaluatorResult.t) =
        switch (dr) {
          | BoxedValue(d) =>
            let (hci, d) = pp_eval(hci, d, (x, parent_hc));
            (hci, BoxedValue(d));
          | Indet(d) =>
            let (hci, d) = pp_eval(hci, d, (x, parent_hc));
            (hci, Indet(d));
          };
          (hci, VarBstMap.add(x, dr, new_env));
        },
        sigma >| EvalEnv.result_map_of_evalenv,
        (hci, VarBstMap.empty),
      );
    (hci, (ei, result_map));
  );
}

/* Postprocessing driver.

See also HoleClosureInfo.res[]/HoleClosureInfo_.rei.
*/
let postprocess = (d: DHExp.t): (HoleClosureInfo.t, DHExp.t) => {
  let (hci, d) =
    pp_eval(HoleClosureInfo_.empty, d, ("", HoleClosure.result_hc));
  (hci >| HoleClosureInfo_.to_hole_closure_info, d);
};

}

```

Listing 19: EvalPostprocess.re

```
/* Program.re */
/* (...)

let get_result = (program: t): Result.t => {
  switch (program) > get_elaboration > evaluate) {
  | (es, BoxedValue(d)) =>
    let (hci, d_postprocessed) =
      switch (d) > EvalPostprocess.postprocess) {
      | d => d
      | exception (EvalPostprocessError.Exception(reason)) =>
        raise(PostprocessError(reason))
    };
    Result.mk(BoxedValue(d_postprocessed), d, hci, es);
  | (es, Indet(d)) =>
    let (hci, d_postprocessed) =
      switch (d) > EvalPostprocess.postprocess) {
      | d => d
      | exception (EvalPostprocessError.Exception(reason)) =>
        raise(PostprocessError(reason))
    };
    Result.mk(BoxedValue(d_postprocessed), d, hci, es);
  | exception (EvaluatorError.Exception(reason)) => raise(EvalError(reason))
  };
/* (...) */
```

Listing 20: Program.re

```
/* Result.rei */
/* The result of a program evaluation.

Components:
- 'EvaluatorResult.t': (postprocessed) evaluation result
- 'HoleClosureInfo.t': information about the holes[hole
closure numbers (from postprocessing)
- 'DHEzp.t': un-postprocessed evaluation result (start point for
evaluation of fill-and-resume)
- 'Delta.t': hole context info (from elaboration, for fill-and-resume)
- 'EvalState.t': evaluation state (for fill-and-resume)
*/
[@deriving sexp]
type t;
```

```

/* First argument is the postprocessed result, second argument is
   the un-postprocessed result (raw evaluation result). */
let mk: (EvaluatorResult.t, DHExp.t, HoleClosureInfo.t, EvalState.t) => t;

/* For displaying the result */
let get_dhexp: t => DHExp.t;
let get_result: t => EvaluatorResult.t;
let get_hci: t => HoleClosureInfo.t;

/* For full-and-resume[0] continuing evaluation */
let get_eval_state: t => EvalState.t;
let get_unpostprocessed_dhexp: t => DHExp.t;

/* See DHExp.fast_equals. Also checks that all environments
   in the HoleClosureInfo.t are equal. */
let fast_equals: (t, t) => bool;

```

Listing 21: Result.rei

```

/* Result.rei */

[@deriving sexp]
type t = (EvaluatorResult.t, DHExp.t, HoleClosureInfo.t, EvalState.t);

let mk =
  (
    dr_result: EvaluatorResult.t,
    d_unpostprocessed: DHExp.t,
    hci: HoleClosureInfo.t,
    es: EvalState.t,
  )
  : t => (
  dr_result,
  d_unpostprocessed,
  hci,
  es,
);

let get_result = ((d, _, _, _): t) => d;
let get_dhexp = (r: t) =>
  switch (r > get_result) {
  | BoxedValue(d)
  | Indet(d) => d
  | _;
  };
let get_hci = ((_, _, hci, _): t) => hci;
let get_eval_state = ((_, _, _, es): t) => es;

```

```

let get_unpostprocessed_dhexp = ((_, d_unpostprocessed, _, _): t) => d_unpostprocessed;
let final_dhexp_equals = (r1: EvaluatorResult.t, r2: EvaluatorResult.t): bool => {
  switch (r1, r2) {
    | (BoxedValue(d1), BoxedValue(d2))
    | (Indet(d1), Indet(d2)) => DHExp.fast_equals(d1, d2)
    | _ => false
  };
};

let fast_equals = ((r1, _, hci1, _): t, (r2, _, hci2, _): t): bool => {
  /* Check that HoleClosure instances are equal */
  MetaVarMap.cardinal(hci1) == MetaVarMap.cardinal(hci2)
  && List.for_all2(
    /* Check that all holes are equal */
    ((u1, hci1), (u2, hci2)) =>
    u1 == u2
    && List.length(hci1) == List.length(hci2)
    && List.for_all2(
      /* Check that all hole closures are equal */
      ((sigma1, _), (sigma2, _)) =>
      EvalEnv.id_of_evalenv(sigma1)
      == EvalEnv.id_of_evalenv(sigma2)
      && List.for_all2(
        /* Check that variable mappings in evalenv are equal */
        ((x1, r1), (x2, r2)) =>
        x1 == x2 && final_dhexp_equals(r1, r2),
        EvalEnv.alist_of_evalenv(sigma1),
        EvalEnv.alist_of_evalenv(sigma2),
      ),
      hci1,
      hci2,
    ),
    MetaVarMap.bindings(hci1),
    MetaVarMap.bindings(hci2),
  )
  /* Check that r1, r2 are equal */
  && final_dhexp_equals(r1, r2);
};

```

Listing 22: Result.re

C.2.5 Unique hole closures

```

/* MetaVar.rei */
/* A "MetaVar.t" represents a hole number 'u'.
   The name "metavar" comes from CMTT. */
[Deriving sexp]

```

```
/* MetaVar.rei */
open Sexplib.Std;
[@deriving sexp]
type t = int;
let eq : (t, t) -> bool;
```

Listing 23: MetaVar.rei

```
/* MetaVar.re
open Sexplib.Std;
[@deriving sexp]
type t = int;
let eq = (x: t, y: t) => x === y;
```

Listing 24: MetaVar.re

```
/* HoleClosureId.rei */
/* Identifier for a hole closure (unique among
   hole closures for a given hole number)
 */
[@deriving sexp]
type t = int;
```

Listing 25: HoleClosureId.rei

```
/* HoleClosureId.re
open Sexplib.Std;
[@deriving sexp]
type t = int;
```

Listing 26: HoleClosureId.re

```
/* HoleClosure.rei */
/* Representation of a hole closure (the set of hole
   instances with the same hole number and environment)
```

```

/* Replacement for HoleInstance.t (due to performance issue,
   should group instances with the same environment together)
*/
[@deriving sexp]
type t = (MetaVar.t, HoleClosureId.t);

let u_of_hc t => MetaVar.t;
let i_of_hc t => HoleClosureId.t;

/* Special HoleClosure.t used to represent the parent
   "hole instance" of the result. That is to say, if a hole
   instance has this value as its parent, then it is
   directly in the result.
*/
let result_hc: t;

```

Listing 27: HoleClosure.rei

```

/* HoleClosure.re */
open Sexplib.Std;
[@deriving sexp]
type t = int;

```

Listing 28: HoleClosure.re

```

/* HoleClosureInfo_.rei */
/* Auxiliary data structure for constructing a
   HoleClosureInfo.t. Useful for building the HoleClosureInfo,
   because we can index it by EvalEnvId. However,
   when using it we want sequential numbers (HoleClosureId)
   to identify the hole closures (similar to HoleInstanceState.t).
*/
[@deriving sexp]
type t =
  MetaVarMap.t(
    EvalEnvIdMap.t((HoleClosureId.t, EvalEnv.t, HoleClosureParents.t)),
  );
let empty: t;
/* The result type for function 'get_hc_id', which is similar to the
   'next' function in HoleInstanceState. If the given closure exists
   in type t, the result also includes the EvalEnv.t stored; otherwise,
   it's a new closure and no EvalEnv shall be returned.

```

```

/*
type hc_id_result =
  ExistClosure(t, HoleClosureId.t, EvalEnv.t)
  | NewClosure(t, HoleClosureId.t);

/* Gets the hole closure id of a hole closure with the given
hole number and hole environment. Also adds the current parent
hole closure to the HoleClosureInfo..t.

If a hole closure with this hole number and environment already
exists in the HoleClosureInfo..t, then the parent hole closure is
added to the HoleClosureInfo..t, and the HoleClosureId.t and
environment of the existing hole closure is returned.

Otherwise, a new HoleClosureId.t is assigned to this hole closure,
and is inserted into the HoleClosureId..t. This is returned, along
with a None EvalEnv.t.

(similar to HoleInstanceInfo.next, but memoized by EvalEnvId.t)
*/
let get_hc_id:
  t, MetaVar.t, EvalEnv.t, HoleClosureParents.t_) => hc_id_result;

/* Updates the environment of the specified hole closure.

(similar to HoleInstanceInfo.update_environment)
*/
let update_hc_env: (t, MetaVar.t, EvalEnv.t) => t;

/* Converts HoleClosureInfo..t to HoleClosureInfo.t */
let to_hole_closure_info: t => HoleClosureInfo.t;

```

Listing 29: HoleClosureInfo..rei

```

/* HoleClosureInfo..rei */

[@deriving sexp]
type t =
  MetaVarMap.t(
    EvalEnvIdMap.t((HoleClosureId.t, EvalEnv.t, HoleClosureParents.t)),
  );

let empty = MetaVarMap.empty;

type hc_id_result =
  ExistClosure(t, HoleClosureId.t, EvalEnv.t)
  | NewClosure(t, HoleClosureId.t);
let get_hc_id =
  (hci: t, u: MetaVar.t, sigma: EvalEnv.t, parent: HoleClosureParents.t_) =>
  hc_id_result => {
    let ei = sigma >| EvalEnv.id_of_evalenv;

```

```

switch (hci |> MetaVarMap.find_opt(u)) {
  /* Hole already exists in the HoleClosureInfo_.t */
  | Some(hcs) =>
    switch (hcs |> EvalEnvIdMap.find_opt(ei)) {
      /* Hole closure already exists in the HoleClosureInfo_.t.
       * Add parent_eid to eids */
      | Some((i, sigma, hole_parents)) =>
          ExistClosure(
            hci
            |> MetaVarMap.add(
              u,
              hcs
              |> EvalEnvIdMap.add(
                ei,
                (
                  i,
                  sigma,
                  parent |> HoleClosureParents.add_parent(hole_parents),
                  ),
                  ),
                  ),
                  i,
                  sigma,
                  )
      /* Hole exists in the HoleClosureInfo_.t but closure doesn't.
       * Create a new hole closure with closure id equal to the number
       * of unique hole closures for the hole. Return a None environment */
      | None =>
        let i = hcs |> EvalEnvIdMap.cardinal;
        NewClosure(
          hci
          |> MetaVarMap.add(
            u,
            hcs
            |> EvalEnvIdMap.add(
              ei,
              (i, sigma, parent |> HoleClosureParents.singleton),
              ),
              i,
              );
            )
    }
  /* Hole doesn't exist in the HoleClosureInfo_.t */
  | None =>
    NewClosure(
      hci
      |> MetaVarMap.add(
        u,
        EvalEnvIdMap.singleton(
          ei,
          (0, sigma, parent |> HoleClosureParents.singleton),
          ),
          ),
          0,
          0,
          );
}

```

```

        )
    );
};

let update_hc_env = (hci: t, u: MetaVar.t, sigma: EvalEnv.t): t => {
    let ei = sigma >> EvalEnv.id_of_evalenv;
    hci
    >> MetaVarMap.update(
        u,
        Option.map(hcs => {
            hcs
            >> EvalEnvIdMap.update(
                ei,
                Option.map(((hcid, _, parents)) => (hcid, sigma, parents)),
                )
            ),
        );
    );
};

let to_hole_closure_info = (hci: t): HoleClosureInfo.t =>
/* For each hole, arrange closures in order of increasing hole
closure id. */
    hci
    >> MetaVarMap.map(
        (
            hcs:
            EvalEnvIdMap.t(
                HoleClosureId.t, EvalEnv.t, HoleClosureParents.t),
            ),
        ) =>
    hcs
    >> EvalEnvIdMap.bindings
    >> List.sort(((i1, _, _), (i2, _, _)) =>
        compare(i1, i2)
    )
    >> List.map(((_, (sigma, hc_parents))) => (sigma, hc_parents))
);

```

Listing 30: HoleClosureInfo.re

```

/* HoleClosureInfo.rei */
/* Stores information about all hole closures reachable
by a program's evaluation result. Used in the context
inspector.

Constructed using HoleClosureInfo.t. */
[@[deriving sexp]]
type t = MetaVarMap.t(List((EvalEnv.t, HoleClosureParents.t)));

let empty: t;

```

```

/* Number of unique closures for a given hole. */
let num_unique_hcs : t, MetaVar.t) => int;

/* Returns the information for a given hole and hole closure
   id, if found. */
let find_hc_opt :
  t, MetaVar.t, HoleClosureId.t) =>
  option((EvalEnv.t, HoleClosureParents.t));

```

Listing 31: HoleClosureInfo.rei

```

/* HoleClosureInfo.re */
open Sexplib.Std;

[@deriving sexp]
type t = MetaVarMap.t(list((EvalEnv.t, HoleClosureParents.t)));

let empty: t = MetaVarMap.empty;

let num_unique_hcs = (hci: t, u: MetaVar.t): int => {
  switch (hci >| MetaVarMap.find_opt(u)) {
    | Some(hcs) => hcs >| List.length
    | None => 0
  };
};

let find_hc_opt =
  (hci: t, u: MetaVar.t, i: HoleClosureId.t) =>
  option((EvalEnv.t, HoleClosureParents.t)) => {
    switch (hci >| MetaVarMap.find_opt(u)) {
      | Some(hcs) => List.nth_opt(hcs, i)
      | None => None
    };
};

```

Listing 32: HoleClosureInfo.re

```

/* HoleClosureParents.rei */
/* List of hole closure parents. Analogous to InstancePath, but a single
   hole closure (set of closures with the same environment) may have
   multiple parents.
*/
[@deriving sexp]

```

```
/* type t_ = (Var.t, HoleClosure.t)
   and t = list(t_); */

let add_parent: (t_, t_) => t;
let to_list: t => list(t_);
let singleton: t_ => t;
```

Listing 33: HoleClosureParents.rei

```
/* HoleClosureParents.re */
open Sexplib.Std;

[@deriving sexp]
type t_ = (Var.t, HoleClosure.t)
and t = list(t_);

let add_parent = (hcp: t, new_parent: t_) => [
  new_parent,
  ...List.filter(p => p != new_parent, hcp),
];
let to_list = (hcp: t): list(t_) => hcp;
let singleton = (parent: t_) => [parent];
```

Listing 34: HoleClosureParents.re

C.2.6 FAR

```
/* FillAndResume.rei */
/* Utilities for fill-and-resume. Fill-and-resume is an alternative
strategy for evaluation that can be performed if all changes in
a recent update happen within a hole in an earlier state, avoiding
re-evaluation of the program from the start. Originally described
in the Hazelnut Live 2019 paper.

See usage in 'Model.update_program'.

*/
/* Perform the fill operation. Fill hole 'u' with the provided
DHExp.t in the result.

fill(exp_to_fill, u, preu_program_result) => filled_program_result

Note that this operates on the UHExp.t level, since hole filling is
```

```

    operationally on that level. */
let fill: (DHExp.t, MetaVar.t, Result.t) => Result.t;

/* Determine if a new program is a filled version of a past program.

is_fill_viable(old_program, new_program) =>
  None if a fill is not viable (perform regular evaluation)
  Some((exp_to_fill, u)) if a fill is viable (perform fill-and-resume)

Essentially performs a structural diff on the programs' elaborated
expressions, and returns a hole if the root of the diff lies in a hole.

See 'DiffDHExp.rei' for more details about the diff-ing process.

TODO: This can be applied on multiple past states. Currently, it is
only applied on the most recent state. This should be a relatively
inexpensive computation, so this should be reasonable. */

let is_fill_viable: (Program.t, Program.t) => option((DHExp.t, MetaVar.t));

```

Listing 35: FillAndResume.rei

```

/* FillAndResume.rei */

/* Preprocesses the previous evaluation result before re-evaluating with
fill-and-resume. Performs two actions:
- Set the 're_eval' flag of 'DHExp.Closure' variants to 'true' so they
will be re-evaluated.
- Substitute holes with the matching hole number.
*/
let rec preprocess = (u: MetaVar.t, d_fill: DHExp.t, d: DHExp.t): DHExp.t => {
  let preprocess: DHExp.t => DHExp.t = preprocess(u, d_fill);

  switch (d) {
    /* Hole types: fill if the hole number matches. */
    | EmptyHole(u', _) =>
    | Keyword(u', _, _) =>
    | FreeVar(u', _, _) =>
    | InvalidText(u', _, _) => u == u' ? d_fill : d
    | NonEmptyHole(reason, u', i, d) =>
      u == u' ? d_fill : NonEmptyHole(reason, u, i, d) |> preprocess
    | InconsistentBranches(u', i, Case(scrut, rules, case_i)) =>
      u == u'
      ? d_fill
      : InconsistentBranches(
          u',
          i,
          Case(
            scrut |> preprocess,
            rules
          ) |> List.map((DHExp.Rule(dp, d)) =>
            DHExp.Rule(dp, d) |> preprocess)
        )
  }
}

```

```

        ),
        case_i,
    ),
)

/* Generalized closures: need to set the `re_eval` flag
   to true, and recurse through the environment.

TODO: memoize environments so we don't need to re-preprocess
the same environment every time it is re-encountered.
*/
Closure[env, _, d] =>
Closure[env
env
|> EvalEnv.map_keep_id(., dr) =>
switch(dr) {
| Indet(d) => Indet(d |> preprocess)
| BoxedValue(d) => BoxedValue(d |> preprocess)
},
),
true,
d |> preprocess,
)

/* Other expressions forms: simply recurse through subexpressions. */
BoolLit(_)
IntLit(_)
FloatLit(_)
BoundVar(_)
Triv => d
Let(dp, d1, d2) => Let(dp, d1 |> preprocess, d2 |> preprocess)
FixF(x, ty, d) => FixF(x, ty, d |> preprocess)
Fun(dp, ty, d) => Fun(dp, ty, d |> preprocess)
Ap(d1, d2) => Ap(d1 |> preprocess, d2 |> preprocess)
ApBuiltin(f, args) => ApBuiltin(f, args |> List.map(preprocess))
BinBoolOp(op, d1, d2) =>
BinIntOp(op, d1, d2) =>
BinFloatOp(op, d1, d2) =>
BinFloatOp(op, d1 |> preprocess, d2 |> preprocess)
ListNil(ty) => d
Cons(d1, d2) => Cons(d1 |> preprocess, d2 |> preprocess)
Inj(ty, side, d) => Inj(ty, side, d |> preprocess)
Pair(d1, d2) => Pair(d1 |> preprocess, d2 |> preprocess)
ConsistentCase(Case(scrut, rules, i)) =>
ConsistentCase(
  Case(
    scrut |> preprocess,
    rules
    |> List.map((DHExp.Rule(dp, d)) => DHExp.Rule(dp, d |> preprocess)),
    i,
  ),
)
Cast(d, ty1, ty2) => Cast(d |> preprocess, ty1, ty2)

```

```

| FailedCast(d, ty1, ty2) => FailedCast(d |> preprocess, ty1, ty2)
| InvalidOperation(d, reason) => InvalidOperation(d |> preprocess, reason)
|_
};

let fill = (d: DHExp.t, u: MetaVar.t, prev_result: Result.t): Result.t => {
  /* Perform FAR preprocessing (fill) */
  let d_result =
    prev_result |> Result.get_unpostprocessed_dhexp |> preprocess(u, d);

  /* Re-start evaluation */
  let (es, dr_result) =
    switch (d_result)
      |> Evaluator.evaluate(
        prev_result |> Result.get_eval_state,
        EvalEnv.empty,
        )
    )
  | (es, dr_result) => (es, dr_result)
  | exception (EvaluatorError.Exception(err)) =>
    raise(Program.EvalError(err))
};

/* Ordinary postprocessing */
let (hci, d_result, dr_postprocessed) =
  switch (dr_result) {
  | Indet(d) =>
    let (hci, d_postprocessed) =
      switch (EvalPostprocess.postprocess(d)) {
      | hci_d => hci_d
      | exception (EvalPostprocessError.Exception(err)) =>
        raise(Program.PostprocessError(err));
      };
    (hci, d, EvaluatorResult.Indet(d_postprocessed));
  | BoxedValue(d) =>
    let (hci, d_postprocessed) =
      switch (EvalPostprocess.postprocess(d)) {
      | hci_d => hci_d
      | exception (EvalPostprocessError.Exception(err)) =>
        raise(Program.PostprocessError(err));
      };
    (hci, d, EvaluatorResult.BoxedValue(d_postprocessed));
  };
  Result.mk(dr_postprocessed, d_result, hci, es);
};

let is_fill_viable =
  (old_prog: Program.t, new_prog: Program.t): option((DHExp.t, MetaVar.t)) => {
  let e1 = old_prog |> Program.get_uhexp;
  let e2 = new_prog |> Program.get_uhexp;

```

```

let elaborate = (e: UHExp.t): DHExp.t => {
  exception DoesNotElaborateError;
  switch (e |> Elaborator_Exp.syn_elab(VarCtx.empty, Delta.empty)) {
    | Elaborator_Exp.ElabResult(d, _, _) => d
    | _ => raise(DoesNotElaborateError)
  };
}

switch (DiffDHExp.diff_dhexp(e1 |> elaborate, e2 |> elaborate)) {
  | NonFillDiff
  | NoDiff => None
  | FillDiff(d, u) => Some((d, u))
}
};
```

Listing 36: FillAndResume.re

```

/* DiffDHExp.rei */

/* Structural diff judgment.
 - NoDiff: internal expressions are the same
 - NonFillDiff: diff is not rooted in a hole
 - FillDiff: diff is rooted in a hole, return fill parameters 'd', 'u'
 */
[@deriving sexp]
type t =
  | NoDiff
  | NonFillDiff
  | FillDiff(DHExp.t, MetaVar.t);

/* Compare two `DHExp.t`'s, returning a difference judgment of type `t`. */
let diff_dhexp: (DHExp.t, DHExp.t) => t;
```

Listing 37: DiffDHExp.rei

```

/* DiffDHExp.rei */

type error =
  | ClosureInUnevaluatedExp
  | DiffNotImplemented;

exception Exception(error);

[@deriving sexp]
type t =
  | NoDiff
```

```

| NonFillDiff
| FillDiff(DHExp.t, MetaVar.t);

let get_hole_number_of_dhexp = (d: DHExp.t): option(MetaVar.t) => {
  switch (d) {
    | EmptyHole(u, _, _, _)
    | NonEmptyHole(_, u, _, _)
    | Keyword(u, _, _)
    | FreeVar(u, _, _)
    | InvalidText(u, _, _) => Some(u)
    | _ => None
  };
};

let rec diff_dhexp = (d1: DHExp.t, d2: DHExp.t): t => {
  /* First, we compare two expressions of the same form, to see if the current
   * node should be the diff root. If the expressions are not of the same form,
   * we check if `d1` is a hole expression. */
  switch (d1, d2) {
    /* Closures should not appear in elaborated, non-evaluated result.
     * Don't technically need to throw an error since this should never come up.
     * But here just in case. */
    | (Closure(_, _), _)
    | (_, Closure(_)) => raise(Exception(ClosureInUnevaluatedExp))

    /* Diffing leaf expressions: if different, this is the (non-fill)
     * root of the diff */
    | (Triv, Triv) => NoDiff
    | (BoolLit(_, _), BoolLit(_))
    | (IntLit(_, _), IntLit(_))
    | (FloatLit(_, _), FloatLit(_))
    | (BoundVar(_), BoundVar(_))
    | (ListHil(_), ListNil(_)) => d1 == d2 ? NoDiff : NonFillDiff

    /* Diffing expressions with one subexpression:
     * - If current node is diff, then current node is diff root.
     * - Else use child's diff. */
    | (Inj(ty1, side1, d1'), Inj(ty2, side2, d2')) =>
      ty1 != ty2 || side1 != side2 ? NonFillDiff : diff_dhexp(d1', d2')
    | (FixF(x1, ty1, d1'), FixF(x2, ty2, d2')) =>
      x1 != x2 || ty1 != ty2 ? NonFillDiff : diff_dhexp(d1', d2')
    | (Fun(dp1, ty1, d1'), Fun(dp2, ty2, d2')) =>
      dp1 != dp2 || ty1 != ty2 ? NonFillDiff : diff_dhexp(d1', d2')
    | (Cast(d1', ty1, ty1'), Cast(d2', ty2, ty2')) =>
      (FailedCast(d1', ty1, ty1'), FailedCast(d2', ty2, ty2')) =>
        ty1 != ty2 || ty1' != ty2' ? NonFillDiff : diff_dhexp(d1', d2')
    | (InvalidOperation(d1', reason1), InvalidOperation(d2', reason2)) =>
      reason1 != reason2 ? NonFillDiff : diff_dhexp(d1', d2')

    /* Diffing expressions with more than one subexpression:
     * - If current node is diff, then current node is diff root.
     * - Else check children (see `diff_children2`)
     */
    | (Let(dp1, d11, d12), Let(dp2, d21, d22)) =>

```

```

d1 != d2 ? NonFillDiff : diff_children2(d11, d12, d21, d22)
| (Ap(d11, d12), Ap(d21, d22))
| (Cons(d11, d12), Cons(d21, d22))
(Pair(d11, d12), Pair(d21, d22)) => diff_children2(d11, d12, d21, d22)
(BinBoolOp(op1, d11, d12), BinBoolOp(op2, d21, d22)) =>
op1 != op2 ? NonFillDiff : diff_children2(d11, d12, d21, d22)
(BinIntOp(op1, d11, d12), BinIntOp(op2, d21, d22)) =>
op1 != op2 ? NonFillDiff : diff_children2(d11, d12, d21, d22)
(BinFloatOp(op1, d11, d12), BinFloatOp(op2, d21, d22)) =>
op1 != op2 ? NonFillDiff : diff_children2(d11, d12, d21, d22)

/* Expression variants with >2 subexpressions */
| (ApBuiltIn(f1, args1), ApBuiltIn(f2, args2)) =>
f1 != f2 ? NonFillDiff : diff_children(args1, args2)
| (ConsistentCase(case1), ConsistentCase(case2)) =>
diff_case(case1, case2)

/* Diffing hole expressions: if the hole is different or if there is a
non-fill child diff, then this becomes the new root of the diff. */
| (EmptyHole(u, _), EmptyHole(_))
| (Keyword(u, _, _), Keyword(_))
| (FreeVar(u, _, _), FreeVar(_))
| (InvalidText(u, _, _), InvalidText(_)) =>
d1 != d2 ? FillDiff(d2, u) : NoDiff
| (NonEmptyHole(reason1, u1, i1, d1'), NonEmptyHole(reason2, u2, i2, d2')) =>
if (reason1 == reason2 || u1 == u2 || i1 == i2) {
  FillDiff(d2, u1);
} else {
  switch (diff_dhexp(d1', d2')) {
  | NonFillDiff => FillDiff(d2, u1)
  | diff => diff
  };
}
| (
  InconsistentBranches(u1, i1, case1),
  InconsistentBranches(u2, i2, case2),
) =>
if (u1 == u2 || i1 == i2) {
  FillDiff(d2, u1);
} else {
  switch (diff_case(case1, case2)) {
  | NonFillDiff => FillDiff(d2, u1)
  | diff => diff
  };
}

/* If different variants, then necessarily a diff.
Check if 'd1' is a hole <=> fill diff. */
| _ =>
switch (get_hole_number_of_dhexp(d1)) {
| Some(u) => FillDiff(d2, u)
| None => NonFillDiff
}
;
```

```

}

/* Helper for diffing multiple subexpressions:
 - If no children diff, then no diff.
 - Else if multiple children diff, then current node is diff root
   (non-fill-diff).
 - Else carry through child's diff. */
and diff_children = (ds1: list(DHExp.t), ds2: list(DHExp.t)): t => {
  let diffs =
    List.map2(diff_dhexp, ds1, ds2)
    |> List.filter((diff: t) =>
      switch (diff) {
        | NoDiff => false
        | FillDiff(_) => true
        | NonFillDiff => true
      }
    );
  switch (diffs) {
    | [] => NoDiff
    | [diff] => diff
    | _diffs => NonFillDiff
  };
}

/* This function is for the special case of two children. */
and diff_children2 =
  (d1: DHExp.t, d2: DHExp.t, d21: DHExp.t, d22: DHExp.t): t =>
  switch ((diff_dhexp(d1, d21), diff_dhexp(d12, d22))) {
    | (NoDiff, NoDiff) => NoDiff
    | (FillDiff(_), NoDiff) => FillDiff(_)
    | (NoDiff, FillDiff(_)) => diff
    | _ => NonFillDiff
  };

/* Helper for diffing case expressions. If any of the patterns
or the `t`'s are different, then the case is the root. Otherwise,
apply `diff_children` to the scrut and rule bodies. */
and diff_case =
  (
    Case(scrut1, rules1, i1): DHExp.case,
    Case(scrut2, rules2, i2): DHExp.case,
  )
  : t => {
  let dp_of_rule = (Rule(dp, _): DHExp.rule): DHPat.t => dp;
  let d_of_rule = (Rule(_, d): DHExp.rule): DHExp.t => d;
  if (i1 != i2
    || rules1
    |> List.map(dp_of_rule) != (rules2 |> List.map(dp_of_rule))) {
    NonFillDiff;
  } else {
    diff_children(
      [scrut1, ...rules1 |> List.map(d_of_rule)],
      [scrut2, ...rules2 |> List.map(d_of_rule)],
    );
  };
}

```

```
|};
```

Listing 38: DiffDHExp.re

```
/* Model.re */
/* (In 'Model.update_program') */

/* Decide between regular evaluation or fill-and-resume. */
let old_result = Program.get_result(old_program);
let new_result =
  switch (FillAndResume.is_fill_viable(old_program, new_program)) {
    | None => new_program |> Program.get_result
    | Some((e, u)) => old_result |> FillAndResume.fill(e, u)
  };
/* (End 'Model.update_program') */
```

Listing 39: Model.re

C.2.7 Evaluation state

```
/* EvalState.rei */
/* Evaluation state. Used to store information that is threaded
   throughout calls to 'Evaluator.evaluate', such as the environment
   id generator (so that all environment ID's are unique) and
   evaluation statistics.

   All of these functions return the update 'EvalState.t' as the first
   parameter.
*/
[@deriving sexp]
type t;
/* Constructor used when beginning evaluation */
let initial : t;
/* Emits a new and unique 'EvalEnvId.t'. */
let next_evalenvid : t => (t, EvalEnvId.t);
/* Getter for statistics */
let get_stats : t => EvalStats.t;
/* Update number of evaluation steps in statistics. */
```

```
let inc_steps: t => t;
```

Listing 40: EvalState.rei

```
/* EvalState.rei */
[@deriving sexp]
type t = (EvalEnvId.t, EvalStats.t);

/* 'EvalEnvId.empty' is a special value used for the empty
environment at the beginning of evaluation or when resuming
evaluation (fill and resume). */
let initial = (EvalEnvId.empty + 1, EvalState.initial);

let next_evalenvid = ((ei, stats): t): (t, EvalEnvId.t) => (
  ei + 1,
  stats,
  ei,
);
let inc_steps = ((ei, stats): t): t => (ei, stats |> EvalStats.inc_steps);
let get_stats = ((_, stats): t): EvalStats.t => stats;
```

Listing 41: EvalState.re

```
/* EvalStats.rei */
/* Evaluation state. Used to store information that is threaded
throughout calls to 'Evaluator.evaluate', such as the environment
id generator (so that all environment ID's are unique) and
evaluation statistics.

All of these functions return the update 'EvalState.t' as the first
parameter.
*/
[@deriving sexp]
type t;

/* Constructor used when beginning evaluation */
let initial: t;

/* Emits a new and unique 'EvalEnvId.t'. */
let next_evalenvid: t => (t, EvalEnvId.t);

/* Getter for statistics */
let get_stats: t => EvalStats.t;
```

```
/* Update number of evaluation steps in statistics. */
let inc_steps : t => t;
```

Listing 42: EvalStats.rei

```
/* EvalStats.re */
open Sexplib.Std;
(* Current implementation: store number of evaluation steps. *)
[@deriving sexp]
type t = int;
let initial = 0;
let inc_steps = (steps: t): t => steps + 1;
let get_steps = (steps: t): int => steps;
```

Listing 43: EvalStats.re

Jonathan Lam
Prof. Barrett
Engineering Management
EID-370
5 / 3 / 22

Response to *The Apple Experience* by Carmine Gallo

I particularly liked the presentation on Carmine Gallo's *The Apple Experience: Secrets to Building Insanely Great Customer Loyalty*, as presented by Alex Cho. Personally, I am not a big fan of Apple's products technologically, but I can say that their marketing really works.

Most of the guidelines are things that seem to be common knowledge today, at least among large technological firms. For example, hiring for team fit, and hiring for character rather than purely technical skill are strategies that companies such as Google or Facebook are also well-known for. Thus the employees form a tight-knit, similarly-innovative group, and are more motivated to work.

On the other hand, what makes Apple really stand out is their marketing prowess. For example, the description of Apple's advertisements when compared to IBM's advertisements are striking. Apple is known for the simplicity in its advertising: not only does this make a larger impression, but it also simplifies the process because there are fewer aspects to confuse or offend the target audience. On the other hand, IBM's bloated advertisements were full of information, but they had to be translated to many audiences and had to be screened to make sure that there were no culturally offensive features among multiple target audiences. I would say that many companies nowadays, especially technological companies, have copied Apple's aesthetic with great success.

The other part that really makes Apple stand out is its store environment. It has a very different feel from traditional brick-and-mortar stores: the space was wide-open, employees would circle around so that customers would feel like they were barely waiting ("resetting the customers' 'internal clock'"), and customers were encouraged to interact with products around the store. The atmosphere would feel almost familial, something that wasn't as expected in a business setting.

In other words, the Apple way of selling was less about what the customer feels, about relationships rather than the product itself. We can also tie this into the previous point about making sure to hire employees by fit and making sure to enrich the employees' lives as well as the customers': this helps build more passionate relationships between employees and customers.

What I would have liked to see more out of in this book is the science behind these marketing techniques. I'm sure there is some mix of intuition or informal experience that informs a lot of these "secrets" of Apple, but I would also assume that there are a lot of user studies and a large body of psychological science that these secrets are based off of. And even if it may not have been a big part of Apple's methodologies in 2012 when the book was written, there's no doubt that Apple is doing user studies nowadays, in the age of big data and machine learning.

I think Gallo presents a lot of interesting ideas that are fascinating in that they work and they are things that we take for granted, perhaps like how we take for granted many human relationships. While I may not be able to apply much of this as a more technical engineer rather than a salesperson, I still believe that parts of this will apply. In particular, the hiring process and the idea of enriching the employees' lives will probably also carry over to engineers like me. Another way I think this book helps is by promoting a general mindfulness: pay attention to the customer, what they're doing, what they're feeling, what they desire, and what they are lacking; by paying attention to the things you might miss if you treat the interaction as a purely business relationship, you may be able to get a much more fulfilling (and profitable) transaction with another person.

Jonathan Lam
Prof. Barrett
EID370
Engineering Management
5 / 10 / 22

Lessons on engineering management and applications to software engineering

At some point, I looked down on managers and people studying management. To me, managing is purely a soft skill that could be easily picked up, and I thought that only the brilliant mathematicians and scientists and engineers who were working at the pinnacle of their respective fields were the smart ones. Through this class, though, I learned that management is a skill that can be taught and learned, and is not intuitive or trivial at all.

It seems to me that engineering management is a fairly intuitive process, but that it may not be easy to tell from a glance what methods work from what methods don't work. As I had thought, management is a role for people with elevated soft skills, but this is far from easy due to the inconstant and sensitive balance of every individual. It also seems that many other engineers have fallen into a similar trap of looking down on management: we learned that many engineers (or people in other technical roles) who have risen to the role of engineering management initially end up stronghanding their position (e.g., via micromanagement), since they have a preconception that management is solely about leading and owning the group of subordinates.

No, we learned that management is not about leading or owning or controlling people; management is about *organizing a group of people towards achieving the company goals*. Anything else is secondary and an intermediate goal towards the company goals. One may consider additional characteristics such as respecting employees or promoting efficiency as part of the definition for management, but really these are still subordinate to the more general definition provided above.

I found the evolution of engineering management thought to be extremely formative. There was not really a strong need for management before the Industrial Revolution, because there are not many large organizations of people working together for extended periods of time, except perhaps outside of

the military or government. Thus military management was some of the oldest forms of management. We did not study ancient military management because it may be too different and outdated for modern engineering management, even if there are probably many overlapping traits.

In the Industrial age, we see major factors contributing to the need for management. Firstly, mass production and efficiency were the new forces behind innovation and human progress. Secondly, people were moving away from traditional tradesman jobs, which work at a much smaller scale than industrial factories and with much fewer people in tandem. Lastly, there is a change in the nature of many of the jobs, especially with the use of machines – many jobs have become less “human” and more repetitive. The question of how to keep these large groups of people working together efficiently and effectively, while not becoming demotivated, becomes a very important logistical problem.

Enter some of our classical schools of thought. We have Frederick Taylor, who was one of the pioneers in scientific management in the late 19th century. Taylor’s methods were solely aimed at improving efficiency in the steel-making processes at his firms. He proposed many logical improvements to the current processes, such as incentivizing above-standard work with extra pay, training new workers, rest breaks, and division of work between workers and management. While these ideas may be common sense for us nowadays and more or less still valid today, they were not in widespread practice before then. The Gilbreths also sought to improve efficiency through the use of motion studies. These methods, which systematically assess actions to improve efficiency to the utmost, are similar to data-driven methods today.

Taylor, however, was often hated by his workers because he made them perform more work. Around the same time, the idea of behavioral management (or “human relations management”) arose, most notably with the Hawthorne Experiments by Elton Mayo at the Western Electric Company. The experiments arose out of a desire to improve efficiency, but their findings were unexpected. The first experiment, with the six workers and changes to the work environment, found that workers tend to do better if consulted by management about changes to their environment, irrelevant of the actual change.

The second (and less famous) experiment showed that certain changes to improve efficiency may always be rejected by an existing group, and thus that existing group norms must be taken into consideration when implementing any change. Together, these experiments show the importance of understanding that the needs of the group and the individual need to be considered, and that the human psyche is not straightforward, predictable, or intuitive. While Taylor thought that everything could be solved with plain "common sense," Mayo showed that this is not the case. Undoubtedly, the Hawthorne effect is widely accepted today, and feedback for management changes are commonly implemented today.

At some point, with the introduction of many different management styles, some of the ideas began to consolidate into a set of general principles. Henry Fayol's Fourteen Principles are probably the most eminent. These fourteen rules give a general framework for thinking about organizing a large group of people into a hierarchical structure. Briefly summarized, these rules are:

1. *Division of work*: Workers should specialize to a small number of tasks.
2. *Authority and responsibility*: Authority and responsibility should be appropriately delegated so that people can carry out their tasks effectively.
3. *Discipline*: Appropriate levels of discipline are necessary to ensure order in the workplace.
4. *Unity of command*: Each worker should have a single manager.
5. *Unity of direction*: There should be a single overall plan that everyone is working towards.
6. *Subordination of individual interests to the general interests*: The company's goals are foremost, even before employees' own goals.
7. *Remuneration*: Employees should be appropriately paid.
8. *Centralization*: An appropriate level of (de)centralization dependent on the circumstances may increase overall organizational efficiency.
9. *Scalar chain*: Chains of authority should be vertical and not horizontal, to avoid confusion.
10. *Order*: Order should be obtained by establishing well-defined roles for everyone.

11. *Equity*: Each employee should be treated equally and fairly.
12. *Stability of tenure of personnel*: This may help improve performance and lower costs and time for training employees.
13. *Initiative*: Employees should be encouraged to make plans for themselves.
14. *Esprit de corps*: The workplace should be a friendly place that encourages working together.

To me, these set of rules again feel very intuitive, but that may only be because these ideas are so deeply intertwined with modern corporate culture.

With the rise of a new type of machines (the digital computer), there are inevitably another wave of huge changes to the nature of work. Computers became deeply intertwined with work with a new “data-driven management,” in which the scientific simulations or machine-learning predictions generated by computers are incorporated into the decision-making process. Data-driven methods are the driving force of many successful technology companies today, so the application of data-driven computer methods to management is no surprise.

As the number of management styles grew, people inevitably came to realize that the circumstances dictate the style of management. This itself may be called another style of management, called the “contingency approach.” This is represented by the black-box in Barrett’s model, and shows that we must be agile to the scenario. On a similar vein, we also realize that all styles of management require an understanding of the entire organization as a complex system of many entities and interactions, and thus we may picture management abstractly using the “systems approach.”

In sum, the whole of management history comes down to “it depends.” This shows how difficult management is, and how far off I originally was when looking down on management. It seems that we can only build up a repertoire of general management ideas, and apply the most relevant ones.

In my field of work, which is software engineering, the most common form of management is a fairly standard functional organization. There are “levels” of managers (called simply “Level 1,” “Level 2,” and on, which higher levels corresponding to higher levels of management). Perhaps the

most widespread management theory in software engineering is something called the Agile methodology, which encourages rapid feedback through iterative development¹. The smallest unit of management in software engineering is usually called the Scrum team, where Scrum is a specific set of actions that coexists with the Agile methodology. We also clearly see many of the other methodologies that were studied throughout the book, from Taylor's idea of training workers (which corresponds to an "onboarding" or "bootcamp" process), to Gilbreth's data-driven efficiency, to Mayo's feedback-driven management, to Fayol's "esprit de corps" with comfy workplaces, to the Japanese management's Kanban method of just-in-time development (Kanban boards are common in software development, e.g., through the Jira system).

I would best describe the software engineering approach as being a highly data-driven contingency approach. As the name suggests, the Agile technology, with its short development cycles and rapid feedback, allows for software companies to be extremely fast-moving and worker-friendly. Since it is data-driven, the things that work best (through past historical data) may be used to influence future decisions, even if they are perhaps not conventional wisdom, and rules can be broken if it improves efficiency and does not hurt overall management. For example, I believe that there is a lot of horizontal communication between employees for various reasons, which encourages overall knowledge share and camaraderie; this is in contrast to scalar chain, which specifies a more rigid structure in which only vertical communication is recommended.

The debates over management styles are becoming ever more relevant nowadays as the COVID-19 pandemic and work-at-home policies prompt another great impetus for changing work and management styles². There was the so-called "Great Resignation" movement during the pandemic due to loss of motivation and disillusionment with work. While we must put some blame on the pandemic

¹ Honestly, I'm surprised this didn't come up in class, given how pervasive the idea is in software engineering. Perhaps it is because it is more recent than many of the topics in this class – many of the recommended books on software giants such as Apple, Google, and Amazon date back two decades, but the Agile methodology has only really become dominant in the last decade or so. It would be good to have some modern books on management in software engineering for this course.

² I believe that management since COVID-19 would be another great book topic for future classes.

and the inevitable stay-at-home and remote work policies, this is also an important challenge for managers to be able to adapt to such situations, continuing to respect the worker, and finding ways to motivate them in the face of greater troubles than ever before.

Luckily, software development companies have weathered the Great Resignation fairly well, due to a number of factors (e.g., software booming during stay-at-home policies, remote work usually works fine for software, Agile methodologies quickly adapted to new policies, etc.) but many other industries (e.g., restaurant workers or medical workers) didn't fare as well. Being able to apply the lessons learned from (engineering) management will be important for any corporation in this volatile time.

Jonathan Lam
Prof. Barrett
Engineering Management
EID-370
2/1/22

Functional Model of Management Applied to my Life

Overview: I choose to plan out the final semester of my time at Cooper. The timeframe is very explicit (up until grades are due at the end of the semester), and this clearly fits within any larger goal of success in career and beyond (which are contingent upon graduating in a timely manner). This is a good way to lay out what I need to plan anyways for my Master's, since there's a lot going on between it and the rest of my school activities.

1. Plan.

1.1. Who/what. The benefactor of this plan, as well as the organizer of the plan, is myself. Additional benefactors include my family members, my thesis advisors, and the people (programmers) who my research groups targets, although these do not really factor much into the planning of my semester.

1.2. Objectives.

The overall objective is attaining a good degree of time management (via delegation, parallelism, and not wasting time) in order to graduate successfully and on time. This can be broken into the following subtasks:

- Meeting the graduation requirements for the undergraduate engineering degree. This mostly involves passing required classes, since all classes have been planned out by this time.
- Meeting requirements of the dual-degree (four year Master's) at Cooper. This is due to my further employment being contingent on the fact that I finish this as planned, and not any later so that I do not have to delay my start date for work.
- Pertaining to the two objectives above, there are many smaller deadlines to meet. For the undergraduate degree, this involves meeting regular deadlines for classes. For the Master's degree, there are the thesis report and presentation deadlines, as well as ensuring that the thesis project meets the minimum requirements in terms of academic rigor.
- Maintaining my other group activities not essential to the bachelor's degree or Master's, which include the "study group" at our research group; weekly shifts at the CUCC; sporadic CS tutoring by student request; and a teaching arrangement for the MATLAB seminar.
- Maintaining good mental and physical health through other activities, such as sleep, eating, maintaining COVID precautions, etc.

1.3. Deadlines.

To better manage deadlines, especially for Master's (in which deadlines are more-or-less managed by myself, rather than in classes where deadlines are managed by the professor), a planning document has been created on Google Drive with a list of weekly goals for the thesis project. This document has been shared with all people relevant to my thesis project.

2. Organize.

2.1. What I have.

- A list of classes and group activities with a set schedule for undergraduate classes and non-essential group activities.
- The preparation from last semester for my Master's project and the ongoing senior capstone project.
- Some job security following Cooper, so I don't have to worry about job applications this semester.
- A close residence to the Cooper Union building, which minimizes transportation time and cost every day.

2.2. What I need to accomplish objectives.

I am not in need of any physical resources to achieve my goal (at the current moment). I have all the resources (time, computing power, access to research group and thesis advisors) necessary to complete my thesis and undergraduate degree, and hard work is all that's remaining to achieve it. This may change over the course of the semester, e.g., if I need access to a paywalled research report.

2.3. How I will get these things.

As mentioned previously, I am not in need of resources.

3. Staff and Direct.

Of course, the most important person in these plans is the organizer, myself. If this person is badly managed or otherwise incapacitated, everything will fall apart.

The people that will be important for the completion of the undergraduate degree mostly involve group members (e.g., for senior capstone projects) and professors. The professors will direct myself in classes, and work on group projects will be distributed between group members and myself.

The people that will be important for the completion of the graduate degree include my thesis advisors (logistical and technical advisors) and another student who is working on very similar material to me. This involves scheduling regular meetings with all of the relevant parties: mostly bi-weekly check-ins.

The people that will be very important for mental health include family members at home, who do most of the cooking and house chores.

Since all of the people that will be essential in completing the Master's already have been "hired," there is no need to appoint new people to these positions.

4. Control.

Using the document containing planning information, I will periodically check back to see if I am on track to finish the planned goals on time. If there are doubts about finishing the Master's project on time, some time may need to be reallocated away from the non-essential group activities into the Master's project.

Additional feedback from thesis advisors or the fellow student working alongside my Master's project will also be useful in reorienting my plans.

Any circumstantial changes to plans (e.g., contracting COVID-19, power outages due to poor weather, etc.) will potentially cause me to also reallocate time towards the Master's project.

Jonathan Lam
Prof. Barrett
EID-370
Engineering Management
4 / 26 / 22

Response to *7 Habits of Highly Effective People*

I was struck by Thodori Kapouranis' explanation of *7 Habits of Highly Effective People* by Stephen R. Covey. Honestly, I tend to avoid books with titles such as these, as I find them to be overly generic and difficult to apply in life. I don't think that this book is much different – the provided advice is still very generic, but the words are fairly striking. They make a lot of sense, and thus resonate with me. Whether or not I will find the willpower to implement these changes in my life will be a different story, but I hope I will be able to in the coming years. Covey promotes seven "habits," which I will review below and discuss why they are relevant to my life:

1. **Be proactive.** I pretty much suck at this. My life has thus far been dominated by deadlines, and in particular school deadlines. There is little incentive (at least for grades) to be proactive, and many students, including myself, fall into the trap of not moving until the last moment possible. Worse, it's easy to rationalize distracting behaviors in various ways; video games may be considered a sort of mental health break, but we may choose to take other kinds of breaks that may relieve the stress from school while also improving ourselves in other ways. As Thodoris states, "we have the freedom to choose our responses based on our self-awareness, conscience, will, and the predicted outcomes of our actions" – we need to choose actions based on what is most important.
2. **Begin with the end in mind.** The big picture. Realizing that a current failure is, in the long run, nothing compared to the more important things in life. That we can move on despite getting a rejection from a college, job, or romantic partner. Or realizing that a particular venture that you worked so hard on helped you but is not worth pursuing further due to diminishing returns or such. For example, I have believed up until this point that my studies have been the most

important investment of my time, but am starting to realize that this is only a step towards greater happiness and enlightenment and social good.

3. **Put first things first.** This habit is the one that stuck with me most. It has a lot to do with the first two habits, and is illustrated by a two-axis table: urgent/not urgent, and important/not important. To be most effective, we need to shift from doing urgent and not important activities to urgent and important activities. If the big picture is in mind, we won't mind that some of the not important activities are unfulfilled. Unfortunately, with schoolwork bombarding us with assignments, each of them seeming more important than the last, it's very easy to become caught up with things that are unimportant in the long run.
4. **Think win-win.** I really like this habit. This is a way to become a better person to others in a single step. It speaks about compromise and the fact that what may benefit one person may also benefit others, and that we should always be striving towards these kinds of solutions. However, as Thodoris notes, this may not always work, such as in competitive environments like sports.
5. **Seek first to understand, then to be understood.** I think I already do this to some extent in an academic context, although I am not as good at empathizing with others. It is always important to be able to understand the other side before attempting to impose your views on others. Often, you will hear interesting perspectives that were foreign and interesting to you.
6. **Synergize.** I feel that this one goes hand-in-hand with "think win-win."
7. **Sharpen the saw.** This is something that my parents have been trying to drill into me since the beginning. Well-roundedness is an important life skill because it will get you through the tough obstacles along the way. Even if you are in a highly intellectual environment, keeping up your physical health is important as a basic requirement, and vice versa. The same is true with spiritual and social needs. In a way, this is like fulfilling the basic needs of Maslow's hierarchy of needs triangle.

Jonathan Lam
Prof. Barrett
EID 370
Engineering Management
02/08/2022

Thoughts on the History of Management Thought

My favorite part of the history of management thought is the Hawthorne experiments by Elton Mayo at the Western Electric Company. I think it's interesting how the studies originally took a very scientific approach, and ended up with a very behavioral (i.e., almost non-scientific) conclusion. It is perhaps even more "scientific" than the observational studies in Taylor's experiments in scientific management, which did not seem to have as clearly-defined of a research process with clear hypothesis testing.

To briefly summarize the event, the Western Electric Company was attempting to improve the productivity of their workers by making changes to the workers' environments and initiating incentives for the workers. The first stage was as described in class: a group of assembly-line workers were separated from the rest of the group, and experiments were performed on various aspects of their work benches such as lighting and rest pauses. However, no matter what changes were performed, it seemed that productivity would always go up. This experiment continued for five years, during which time production had increased from 2,400 to 3,000 relays per day (a 25% increase).

The important thing to note was that this group of workers was consulted before each change, and they had the ability to voice their opinions and sometimes reject suggestions. The company creatively intuited that perhaps the wholehearted participation and cooperation of the whole group, due to the company recognizing the importance of individuals in this group and this social group as a whole in critical decisions about work conditions, will improve worker productivity. This is the famed Hawthorne Effect.

The second part of the experiment adds to the hypothesis that the social norms of a group should be respected. In this experiment, a second group of fourteen workers that performed wiring and

soldering was experimented on in a similar way, but the performance change was the opposite. The conclusion is that due to the wiring team's previously hardwired group norms, such as sticking to an informal production rate cap and ostracizing those who did not follow it. Clearly, this norm would be violated if the production was to go up, and so the group may have felt averse to the experiments. As a result, they were not as cooperative in the experiment, and the results showed.

Two things about the Hawthorne Effect are interesting to take note of. Firstly (and the textbook makes clear point of this), it is important that the company not only listen to individuals, but the social groups formed within a corporation. Secondly, it was only after this experiment that people got a concrete understanding that there is a complex relationship between human factors and productivity, and that productivity is much more than purely mechanical efficiency, as Taylor was studying.

Personally, I also find experimenting with my own workstation for productivity very fun as well: if I have the ability to personalize my work environment rather than being confined to a default work setup, then I generally feel happier and more productive. For example, at my last workplace, even though most people at the company that I was working at use Windows, they had the option to use Linux, which I feel more comfortable with; giving that option of personalization, and also having people listen to my issue with the particular software product on Linux, empowered me to work harder to solve those issues. Similarly, I like being able to personalize my work environment as well, such as using an ergonomic keyboard and keyboard layout; being able to choose computer components for my PC; programming language and editor environment; pen and notebook; monitor resolution, count, and orientation; etc. In the past, I felt that this was a matter of pure efficiency, but I think that there is a mini-Hawthorne Effect here: when I feel that I have the power to change my own work environment, even if not directly or mechanically more efficient, it will improve my motivation to do work.

Jonathan Lam

Prof. Barrett

EID-370

Engineering Management

4 / 5 / 22

On Motivation

In the quiz we had last class about motivation, I got pretty consistent scores. For self-actualization, esteem, social, safety and security, and physiological, I got scores of 4, 8, 8, 4, and 8, respectively. I guess this means that I am pretty evenly and moderately motivated by the different levels of needs.

Personally, I don't know what level of Maslow's hierarchy I lie at. The lines really don't seem very clear cut, but perhaps that is because I am still in school and the overwhelming motivators seem to be related to school. I feel that I have the lower two levels of needs met. I don't feel that my life is in danger, and economically and physically feel somewhat secure. In terms of the upper three levels, it becomes less clear. I feel that my motivation to be socially recognized, to have high self-esteem, and to have focus on self-esteem are more or less on the same level. I like to consider myself as an academic, and I believe that academics have the stereotype of leaving aside social and personal needs in order to advance their intellectual capacity and better humanity (self-fulfillment); however, this is probably a bit idealistic and definitely not entirely true for myself at this stage. I would like to achieve a state where I am so focused on working towards a goal that would enlighten myself while helping others, but I don't think I'm at that state yet.

I think a good way to think of Maslow's hierarchy of needs is to consider the average school project. There is some motivation due to being able to pick project partners: this is a social need to strengthen the bond with friends or network with peers. However, this motivation is not often very great because (at Cooper) there is a small pool of potential teammates and the social or networking

aspect of projects is relatively insignificant. I think esteem plays a bigger part in motivating many school projects. The textbooks defines esteem to be “tied to feelings of achievement, competence, knowledge, maturity, and independence” (455). In general, school projects increase all of these; they increase many skillsets (academic, technical, communication, etc.), expand our knowledge, and simply help us build more experience so we can be more competent and confident. For the most part, the motivation ends here, because of the tough time constraints of schoolwork; we don’t often have the mental bandwidth to try to achieve mental fulfillment, but occasionally there is the chance to really excel and maximize our creative and mental capacity. I believe this is where academics and research tends to strive towards, and this is where I strive towards.

We also discussed in class how sometimes needs have to be reconsidered if lower needs on the pyramid are not met. This happens often with school projects as well: while we are trying to achieve technical competence and maturity (esteem needs), we run into a dozen different deadlines at finals time and then begin to lose sleep and time for hobbies or clubs. This then threatens our lower-level needs: we may even endanger our health in order to keep up satisfactorily with school. For a short term, this may be acceptable and we may override our lower-level needs in order to power through a tough time at school, but a prolonged period will force a student to reconsider their classes. Then they may have to drop a class or settle for less in a project, lowering the higher-level needs in order to make sure the lower-level needs are set.

I feel that the above tradeoff is the same for anything when we talk about time management. Time management is all about managing our needs; in the end, we only have a limited number of hours in the day, and our health (physiological needs) will suffer and force higher-level needs to be diminished in order to simply survive.

Jonathan Lam
Prof. Barrett
EID-370
Engineering Management
4 / 12 / 22

The Motivation to Work

The Motivation to Work is a study by Frederick Herzberg, Bernard Mausner, and Barbara Bloch

Snydermann performed in 1959, aimed at answering the question: "What does the worker want from his job?" Previous studies did not answer this satisfactorily, and did not provide a practical theory.

Herzberg suggests studying F-A-E (factors, attitudes, effects). In this way, we can understand both what factors affect job attitudes, and what effects are caused by job attitudes. To do this, Herzberg asks participants of the study:

"Start with any story you like – either a time when you felt exceptionally good or a time you felt exceptionally bad about your job, either a long-range sequence of events or a short-range one"

The participant may give a response, such as:

"I was promised a pay raise and it didn't come through one pay period after another for three pay periods. I was extremely unhappy even when I received the pay raise because I felt that I should have been given the pay raise when it was promised or at least information about why it was not forthcoming."

In this case, the factor is salary (lack of pay raise). The attitude is a negative job attitude. The effect is a change in personal feelings about the profession, and perhaps the worsening of an interpersonal relationship with a manager that produced the raise.

Herzberg collected many such stories from 203 accountants and engineers from 9 industrial firms in Pittsburgh. The most important results are summarized in the table below.

TABLE 6
Percentage of Each First-Level Factor Appearing in High and Low Job-Attitude Sequences

	Duration of Feelings					
	High			Low		
	Long *	Short	Total	Long *	Short	Total
1. Achievement	38	54	41 †	6	10	7
2. Recognition	27	64	33 †	11	38	18
3. Work itself	31	3	26 †	18	4	14
4. Responsibility	28	0	23 †	6	4	6
5. Advancement	23	3	20 †	14	6	11
6. Salary	15	13	15	21	8	17
7. Possibility of growth	7	0	6	11	3	8
8. Interpersonal relations—subordinate	6	3	6	1	8	3
9. Status	5	3	4	6	1	4
10. Interpersonal relations—superior	4	5	4	18	10	15 †
11. Interpersonal relations—peers	4	0	3	7	10	8 †
12. Supervision-technical	3	0	3	23	13	20 †
13. Company policy and administration	3	0	3	37	18	31 †
14. Working conditions	1	0	1	12	8	11 †
15. Personal life	1	0	1	8	7	6 †
16. Job security	1	0	1	2	0	1

* The Long column includes the frequency of lasting attitudes resulting from both long-range and short-range sequences.

† Differences of totals between high and low statistically significant at .01 level of confidence.

We observe *intrinsic factors* related to the work performed, such as achievement, recognition, work itself, and responsibility, and advancement, were most highly correlated with job satisfaction. We call these *motivators*. On the other hand, *extrinsic factors* related to the job environment, such as company policy and administration, supervision, and salary, tend to be correlated with job dissatisfaction. We call these *hygiene* factors. The word "hygiene" refers to the idea of medical hygiene, which "operators to remove health hazards from the environment of man. It is not a curative; it is, rather, a preventive ... Improvement in these factors of hygiene will serve to remove the impediments to positive job attitude."

This forms the *motivator-hygiene theory*: factors tend to fall into two groups. Satisfying motivators tends to lead to job satisfaction, but not satisfying them doesn't lead to dissatisfaction. Not satisfying hygiene factors leads to dissatisfaction, but satisfying them doesn't lead to motivation.

Jonathan Lam
Prof. Barrett
EID-370
Engineering Management
4 / 12 / 22

Book: *The Motivation to Work*, by Frederick Herzberg, Bernard Mausner, Barbara Bloch Snydermann
Originally published in 1959

Response to *The Motivation to Work*

I decided to read Herzberg's *The Motivation to Work* because I wanted to find out what kinds of mechanisms or guidelines people had developed to keep oneself motivated. As someone who will join the workforce very soon, I think that it will be very important to learn this skill. School and university have always provided a guideline and a set of structured and chronological goals that students have to check off, but work is (from my current perspective) much more unstructured and unprincipled. There is no more structure of school semesters or grades or the distinction between professors and students.

We've just discussed motivation in class. We talked about how self-motivation is the best kind of motivation, but did not talk about ways to self-motivate. In particular, we discussed at length Maslow's hierarchy of needs as a useful model for characterizing the duals of needs and motivations. Lastly, we discussed in brief Herzberg's theory of motivation, which is the subject of this book.

In order to have a useful discussion of the book, I will first provide a summary of *The Motivation to Work*. The work is structured as a research paper, for a study carried out by Herzberg, Mausner, and Snydermann: they state the motivation behind the work and the previous work, describe and justify the experimental methods, discuss a set of two pilot experiments, explain relevant definitions, describe the results, and then discuss implications. The work is very well-written, and it appears to have had wide recognition, as is re-affirmed by the Foreword and Preface sections. The Preface was added in the 1993 edition by Herzberg and discusses the prodigious impact of this study since it was originally published.

The motivation of this study is simple: to find what causes job satisfaction. In other words, the answer to the question, "What does the worker want from his job?" Previous studies were lacking in that they did not provide a useful theory. The end results of this study (called the *motivator-hygiene theory*) will prove to be a practical theory that allows for heuristic implementation. In order to get practical results, Herzberg identified the need to examine F-A-E (factors, attitudes, effects) as a single unit. Previous studies had attempted to find what factors affect job attitudes, or try to ascertain the effect of certain job attitudes, but not the combination of the two.

Herzberg notes that there is an ethical discussion on studying motivation. If companies know what motivates or demotivates workers, this can be used to maliciously manipulate workers. (Peter Drucker was a proponent of this view.) Herzberg argues that, while this is certainly a possibility, companies also have many other ways of manipulating workers, and this study has the ability to provide workers with more fulfilling lives. (I am one of those workers aiming for a more fulfilling life.)

The methodology of the study is heavily discussed throughout the study. As motivation is a difficult thing to measure, Herzberg is very careful with the formulation of the study, and has to assure the reader many times that the method of data-gathering (the *semi-structured interview*) leads to satisfactory results. Luckily, he has a background in such studies, having previously performed research studies such as *Job Attitudes: Review of Research and Opinion*, which was the precursor to this work. In particular, notable researcher John C. Flanagan, director of research at the American Institute for Research, promoted a method of *critical events*, in which study participants are asked to qualitatively describe important events in their career. These qualitative results are then analyzed using a method called *content analysis*, which allows for a partial quantization of the responses so that they may be analyzed. Herzberg follows a very similar approach to Flanagan. The people administering the study are given a series of starter questions. For example, Herzberg gives the following example script:

"Start with any story you like – either a time when you felt exceptionally good or a time you felt exceptionally bad about your job, either a long-range sequence of events or a short-range one"
(35).

Afterwards, the researchers are asked to probe the participants on a number of points, such as the objective events (first-level factors) and the psychological reaction to the events (second-level factors), as well as the criticalness of the events. Thus, from these participants, Herzberg gets a series of stories (*sequences of events*), each with the following features: length (*long-range* or *short-range*), first-level factors, second-level factors, criticalness. These form the datapoints for which to analyze. Two pilot surveys were used to refine this method of questioning.

For the actual study, the population of study is nine industrial companies in Pittsburgh. This comprised 203 interviewees. These interviewees are randomly drawn from the pool of engineers and accountants from these two companies, as Herzberg considered these "two of the most important staff groups in modern industry" (34). The above methods are carried out. Then, to determine factors from the qualitative stories, an *a posteriori* approach to content analysis is used to determine a set of factors from the sequences of events (38).

Once all the data is collected, Herzberg is able to identify a number of factors affecting job attitudes. Notably, given the wording of the question, analysis is split into analysis of "high job attitudes" and "low job attitudes," as it is a key hypothesis of this study that there may be differences between these two. Beginning with high job attitudes, the most common associated factors are achievement, recognition, the work itself, responsibility, and advancement. Each of these are associated with over 20% of positive stories. Other factors that contributed to positive job attitudes but were not as prevalent include job security, personal life, working conditions, company policy, status, possibility of growth, and salary. The key observation here is that the top elements that allow for positive job attitudes are *intrinsic factors*, i.e., ones that relate to the work at hand. On the other hand, the less-

important factors that contribute to positive job conditions are *extrinsic factors*, i.e., relating to the work environment but not the work itself.

Next, we may consider the factors leading to low job attitude. In this case, the relationship tends to be flipped. Extrinsic factors tend to be more affiliated with low job attitude, and intrinsic factors are less correlated. For example, company policy and administration, supervision-technical, and salary were the highest among the associated negative factors. Note that salary may be considered a positive factor, but Herzberg argues that it is more associated with negative factors because when salary alone is a critical factor, it usually means that the person's salary is too low. On the other hand, if salary is increased or leads to positive job attitude, it is usually a byproduct of increased recognition or job advancement, and not the root factor.

So far, we have considered the F-A part of F-A-E. When considering the effects of certain factors (F-E), Herzberg finds an interesting result: if the job attitude is positive, then one tends to experience the same results, irrelevant of what factors cause the positive job attitude. The same is true for negative job attitudes. As a result, effects are largely only dependent on job attitude (an A-E relationship), and not on particular factors (no F-E relationship).

There is a number of other interesting data analyses, but the relationship described above is by far the most prevalent result of this experiment. This forms the foundation of the *motivator-hygiene theory*. This theory states that the factors that tend to cause job satisfaction and the factors that tend to cause job dissatisfaction are largely non-overlapping. Thus the *motivators*, which are largely intrinsic factors related to the work itself, tend to increase job satisfaction. However, not satisfying the motivators will not make the worker very discontented. On the other hand, *hygiene factors*, which are largely extrinsic factors related to the work environment, tend to cause job dissatisfaction when not satisfied. However, satisfying these motivators will not make the worker very satisfied. The word "hygiene" refers to the idea of medical hygiene, which "operators to remove health hazards from the

environment of man. It is not a curative; it is, rather, a preventive ... Improvement in these factors of hygiene will serve to remove the impediments to positive job attitude" (113).

Herzberg compares these to the hierarchy of needs: motivators are related to self-actualization, and hygiene are related to the basic needs. If the basic needs are not met, then one may not be able to feel satisfied, since they are struggling with even economic and regular survival. However, once they are met they serve as a base on which to start achieving job satisfaction. However, to really achieve job satisfaction, one must have the additional case of self-actualization. Without self-actualization, but with basic needs met, one exists in the center layers of the Maslow's triangle, and is neither lacking nor fulfilled.

This mental imagery allows companies to build heuristic models on which to improve job satisfaction. Namely, this comprises the introduction of additional motivators, and/or the removal of poor hygiene factors. Another novel idea proposed by this model is that you cannot put all factors onto a single scale to measure job satisfaction, as there are two sets of unlike factors. Satisfying one of the motivators will have a different effect than satisfying one of the hygiene factors.

We may ask, now, whether this model is still relevant today. And, in particular, whether it is relevant in my field (software engineering). First of all, Herzberg is also sure to take appropriate caution in stating the generality of the results. While the sampled population is that of industrial populations in Pittsburgh, they argue that the diversity in these companies represents a wide variety of professionals in America. However, these are professionals, and in particular accountants and engineers. Both of these jobs involve a mix of intellectual work and standard paperwork. Compare this to front-line service workers or physical laborers, who do not have as many motivators. Thus most of the factors are hygiene factors. And then there are also company officials and leaders, who have primarily intellectual work and can dictate the properties of their environment: these workers primarily have motivator factors. Herzberg states that workers with differing factor profiles such as these may need to be studied separately. However, he notes in the preface that studies following the original work

have shown that the motivator-hygiene theory has stayed remarkably consistent cross-culturally for jobs with a similar factor profile, such as in Japan.

With this in mind, we can start to address the model of modern software engineering, which was a field that did not exist by the time of the original study in 1959. Software engineering is similar to industrial engineering in many ways: it is highly intellectual, but still has the hygiene factors of a typical corporate environment.

We can say that software engineering is a brand new discipline, whose work environments have had the privilege of being shaped by data-driven analytics such as Herzberg's work. One of the distinguishing factors of large software companies today are amazing workspaces with many perks, arguably led by the Googleplex, and followed by other large companies such as Apple Park. These, along with a highly inclusive culture (such as Google's Ten Things¹), are attempts to satisfy any potential general hygiene issues with the workplace. This way, these companies can focus on mental fulfillment by tackling difficult problems. Companies that advertise such exciting perks and wonderful workplaces are willing to invest in tackling hygienes because they know it will pay off. Google probably does many experiments on its workers to discover what benefit lead to the greatest decrease in employee dissatisfaction.

The flip side is to improve motivating factors. With the hygiene factors covered, large companies with nice workplaces can turn to tackling hard and interesting software problems, such as self-driving cars or better data analytics (and the data analytics can help determine motivators and hygienes). I believe that this leads to a virtuous cycle that has led to the blowup of software engineering jobs in these large firms, largely tied to high job satisfaction due to application of Herzberg's principles.

¹ <https://about.google/philosophy/>

Jonathan Lam
Prof. Barrett
Engineering Management
EID-370
2022/02/21

Objective-setting at my previous work experience

I will talk about my experience interning at MathWorks as a software engineer this summer. Simply put, the objectives set forward for me were not very clear, even from the start, so that even now I'm still wondering how much of it was intentional room for growth. At the time, I was in much confusion. I'll begin from the very beginning.

I was hired as part of the general technical development program at MathWorks called EDG. At some point in the interview process, I was asked by a technical hiring manager if I wanted to work on web development, such as MATLAB Online. I said I would be okay with this, but I would prefer if I could work on a different field: web development is where I had the most background work, but I was trying to move into other, more technical fields in CS. This was my first mistake: not clearly communicating my strong preference to move away from web development. With my reluctant agreement to do web development, I was placed into a team that works on the MATLAB front-end.

When I first spoke to my direct manager (team lead), he suggested some amazing ideas about revolutionizing the way programming environments and editors are envisioned. I was amazed, but I also don't think I got much specific detail through that. In the first few weeks of my internship, my goals were all very vague: to get oriented with the editor codebase and find a project.

At the EDG program at MathWorks, there are a number of people that support an intern: the team lead, a technical mentor in the team, an EDG manager (who helps with career questions), and an EDG "buddy" (who helps with general company information and networking). While this is a lot of support, it does cause some level of confusion and violates the "unicity of command" principle by Fayol; I was flitting between reporting to my mentor and to my team lead at the beginning, and trying to set goals with a combination of my team lead and EDG manager, and receiving feedback from all

four. In addition, all four have different goals for the intern: the team lead to facilitate your work within the greater goals of the team; the technical mentor to help the ramp-up process for technical skills; the EDG manager to promote vocational growth and make sure the EDG “final project” is going as planned; and the EDG buddy to ensure that the intern is able to get his way around the company and his general needs met.

In hindsight, I think I was overwhelmed by these goals, and it took me a very long time to get off the ground. I was never set very specific long-term goals by any of these people or myself. We did set short-term goals and discuss progress on them on alternating days, and I did have biweekly meetings with my team lead and EDG mentor. But the only long-term goals were tasks like “finish onboarding tasks and get familiar with the codebase,” or a high-level vision of the project in my team lead’s words. While the latter was very motivational, we rarely got down to specifics; or when we did get to specifics, I would always not entirely understand.

Without the goals being specific, the other aspects of the S.M.A.R.T. principles of objectives fall apart even more completely. Throughout the whole internship, I felt as though I was only moving forward through little steps with little foresight. This also caused my onboarding process to be very long (about two months), which left me alarmingly little time for the final project. While I did manage to pull it off and my team lead appeared to be satisfied with the end result, I felt directionless throughout.

With this hindsight, I will definitely focus more attention on specific objective-setting at my next workplace after graduation. Much of it was my fault for not putting forth the initiative to do so.

Jonathan Lam
Prof. Barrett
EID-370
Engineering Management
4/19/22

Skunk Works Response

The management presentation that I enjoyed most last class was Sam Shersher's presentation about *Skunk Works*, written by Ben Rich. Skunk Works was a small division within Lockheed Martin that was able to produce incredible results given their size and the lack of large funding. While the small size and the dire wartime tensions probably cause this to be an anomalous situation rather than an example of good management practices, it is still a fascinating case study with some lessons to teach.

Much of the focus of Sam's presentation focused around the F-117 Nighthawk stealth fighter. At the time, the US government did not believe that they could get past the USSR's iron curtain and resorted to using ICBM's rather than planes, due to the high failure rate of plane missions. Skunk Works, under the new leadership of Ben Rich, contested the conventional belief under the advice of mathematician and radar specialist, Denys Overholser. The development of the F-117 was very technologically interesting. For example, the theory about radar was based on the 1964 paper "Method of Edge Waves in the Physical Theory of Diffraction" by Soviet mathematician Pyotr Ufimtsev. However, implementing this theory caused issues with the aerodynamic stability of the plane.

The initial development of the planes was under a very tight budget due to the initial skepticism about the anti-radar technology. However, with only a tight \$35 million budget, and in record time¹, Skunk Works was able to produce two demonstrator planes that showed the efficacy of the anti-radar design, and funding was able to significantly increase for stealth planes afterwards. The time and money invested in the F-117 due to the high-risk hypothesis paid off, and the F-117 eventually became Skunk Work's "cash cow."

¹ Goodall, James C. (1992). "The Lockheed F-117A Stealth Fighter". America's Stealth Fighters and Bombers: B-2, F-117, YF-22 and YF-23. St. Paul, MN: Motorbooks International. ISBN 978-0-87938-609-2.

There are a few things to note about the management at play here. First of all, the entire project was based on a risky gamble, based on a scientific claim by an old paper. Luckily, the science was correct, and the technology was advanced enough to solve the aerodynamic stability issues. This somewhat follows the “scientific management” school of management that shows the willingness of managers to counter conventional wisdom using scientific studies. It also shows the importance of quickly generating a proof of concept (PoC) or minimum viable product (MVP) to the larger funding entity (the government), so that the funding and support for such a project will increase.

Of course, since this was essentially a dream team of American engineers and scientists, this is not representative of your average American firm or management style. Also, the team regularly worked 60 hour weeks on this project in order for it to take off. So rather than being an example of what a good management style is, it may be better to say that this is a good example of what is possible with a small talented team that is motivated towards a common goal. It contrasts to the much more organized engineering management styles that we’ve discussed in class. In this case, a bureaucratic system of management would only hamper the rapid development and prototyping process, and a small, matrix-like, close-knit team of specialists is enough to develop the highly advanced product.

In addition to the F-117, Skunk Works was also able to develop the U-2 stealth aircraft and the SR-71 Blackbird, which is still the fastest (non-spacecraft) vehicle. Undoubtedly, this risky management style must have played a large role in allowing for such success.

SS334 Discussion Forum 1

Jonathan Lam

2022/03/27

Technology and the internet has opened the door for a lot of nontraditional goods to study. For example, Bitcoin and other cryptocurrencies may be considered nonexcludable and rival (rival because of the finite size of the Bitcoin pool); privately-owned websites may be considered excludable and nonrival, and access to the Internet itself (which is ultimately tied to public infrastructure, and where bandwidth is almost inconsequential) is nonrival and non-excludable (a public good). Along the same lines, information is a public good, and the Internet is arguably its largest disseminator.

It's interesting that, as the dependence on the Internet grows, the value of networked systems and the data stored on those systems has increased exponentially, and with it the frequency and cost of cyberattacks.

If we briefly consider the U.S.A., there are a wealth of statistics about cybersecurity attacks. For example, IBM and the Ponemon Institute estimate that the average cost of a data breach in the U.S.A. in 2021 was *4.24million, after trillion annually* [1]. This are losses greater than the 3rd largest GDP in the world (Japan, at \$5.06 trillion) [2].

Turning our focus to the Russia-Ukraine conflict, it's clear that proper utilization of the Internet, or the advantages gained by defenses and attacks through the digital medium of networked computers, have very real effects on the outcome of war and on the economy.

For example, take the following cases, reported by law firm Baker Hostetler, which specializes in data issues [3]:

- In 2015, Russian hackers hacked the Ukrainian power grid, causing major outages.
- In 2017, the NotPetya Russian malware aimed at Ukraine's networks caused billions of dollars of damages globally.
- As retaliation against U.S. sanctions, some Russian-based ransomware groups such as Conti have recently posted their victims's data online, even after the victims have paid the ransom.
- The Ukrainian government has asked for volunteer hackers, who have successfully launched some attacks against major Russian websites and against the train systems of Russia's ally Belarus.

- There was a major leak of internal infrastructure and data of a major Russian ransomware group called Conti by pro-Ukraine members, severely jeopardizing the group's activities.

Another example of hacking this one by the hacking group Anonymous has become widely known. A report by security company Security Discovery posted a report of their attempt to fact-check the claims by Anonymous, and they verified 92 compromised Russian databases of governmental websites, multiple hacked Russian state TV stations, and denial-of-service attacks against major Russian state websites such as RT [4].

Other communications-related failures that have a direct impact on military campaigns includes carelessness due to the use of personal phones or unencrypted radios, which have repeatedly caused multiple Russian officials to be killed [5].

These examples probably only represent a tiny fraction of the true extent of the cyberattacks related to this war, many of which will only be declassified or disclosed in the future. Cybersecurity mistakes or attacks have probably caused billions of dollars in both military blunders and in company losses in Russia, Ukraine, and in other nations due to retaliatory measures – and this is on top of traditional war-time costs.

One of the factors that makes information and the Internet particularly interesting are that they are public goods, that any state cannot afford to lose access to. The benefit of cutting a nation's Internet completely off from the rest of the world is that you would get some protection from cybersecurity attacks, but this is far outweighed by the cost of inconveniences for ordinary citizens and by losing free access to highly valuable information. Of course, this doesn't stop nations from attempting to control media, as is famous with China's "Great Firewall" [6], and presumably is similar to what Russia may be attempting now with its state media. As we mentioned in class, privatization is one way to solve issues when there are uncooperative agents with an excludable good. And the Internet is not just full of uncooperative agents; it's full of malicious ones, especially nation states.

The best policy recommendation that I could give is to increase investment in cybersecurity resources and education. The first would be a short-term benefit, and the latter a long-term benefit, both for wartime and peacetime. There are numerous issues with cybersecurity today: while some are fundamentally related to protocols or systems not designed to be secure (e.g., unencrypted email) and are thus "unfixable" without designing new protocols, many cybersecurity issues exist out of human error or even social engineering, which we can train people to protect against.

Bibliography

- [1]: Tunggal, Abi Tyas. "What Is the Cost of a Data Breach in 2021?" Upguard, Upguard, Inc., 23 Feb. 2022, <https://www.upguard.com/blog/cost-of-data-breach>.
- [2]: Oliver, Caleb. "The Top 25 Economies in the World." Investopedia, Investopedia, 4 Feb. 2022, <https://www.investopedia.com/insights/worlds-top-economies/>.
- [3]: Koller, M. Scott. "Impact of the Ukraine/Russia Conflict on Cybersecurity in the United States." Baker Data Counsel, Baker and Hostetler LLP, 16 Mar. 2022, <https://www.bakerdatacounsel.com/data-security/impact-of-the-ukraine-russia-conflict-on-cybersecurity-in-the-united-states/>.
- [4]: Pittrelli, Monica Buchanan. "Anonymous Declared a 'Cyber War' against Russia. Here Are the Results." CNBC, CNBC, 16 Mar. 2022, <https://www.cnbc.com/2022/03/16/what-has-anonymous-done-to-russia-here-are-the-results-.html>.
- [5]: Roth, William, et al. "Russian Generals Are Getting Killed at an Extraordinary Rate." The Washington Post, WP Company, 27 Mar. 2022, <https://www.washingtonpost.com/world/2022/03/26/ukraine-russian-generals-dead/>.
- [6]: Chan, Conrad, et al. "China's Great Firewall." Free speech vs. Maintaining Social Cohesion. Stanford, 2011, https://cs.stanford.edu/people/eroberts/cs181/projects/2010-11/FreeExpressionVsSocialCohesion/china_policy.html.

Jonathan Lam
Prof. Bataille
SS334 Microeconomics
Final Report
5 / 8 / 22

U.S. export sanctions for advanced technologies on China

Preface from the author

The topic of technological trade between the U.S. and China has fascinated me throughout this course. Of course, these are the two largest economies in the world, and my ethnicity is Chinese. China has become a leader in the manufacture and consumption of technology, and a critical party in the discussion of many technological matters, such as cryptocurrency mining or stolen designs (as will be discussed in this paper).

These heated economic topics have far-reaching effects, such as the persecution of Chinese nationals in the United States on false claims of ties to China's Thousand Talents Plan [20]. Although I am not a Chinese national, I am saddened because these claims perpetuate a fear and lack of respect for Chinese people in the United States. The economic issues mentioned in this paper play a great role in that sentiment, especially with the emphasis of the Trump-era export sanctions, but this bigoted sentiment extends far beyond economics. Since I am ethnically Chinese and intend to work in a technological industry, this may affect me as well.

One of my blog post responses and one of my discussion forum responses discusses the relationship between the U.S. and China in tech-related exchanges, so I believe that this topic is a natural consummation of my studies for this course. (N.B. Some of the explanations in this paper are recycled from my old blog post and discussion forum because the topics are relevant.)

Background on relevant (technology) industries and their importance in economics

We examine the trade of advanced technologies between the United States and China, especially as it relates to recent export sanctions imposed by the United States beginning during the Trump presidency and reinforced during the (current) Biden presidency. By "advanced technologies," we are primarily referring to (but not limited to) the foundry and semiconductor chip industries and the communications industry, as these contain the main players of interest in this time period.

First, we briefly discuss international trade in general. Each group or nation has its own comparative advantages in production of certain goods or services. In the case of the United States, the comparative advantage may be in the design of advanced chip designs (such as Apple's in-house M1 designs). In the case of China, the advantage may be in the mass manufacture of smartphones containing those chips. Each nation working in isolation is limited by a maximum efficiency described by the production possibility frontier (PPF). However, a nation may exceed the PPF by means of international trade and focusing production on the goods and services for which it has a comparative advantage. This increases the size of the "economic pie" by benefitting both nations. Thus, it is usually highly desirable for any nation to engage in international trade, except in cases when domestic industries may be hurt (as we will see later in this paper). For such economically powerful nations such as the U.S. and China, there is an enormous benefit to engage in trade with each other, and this is evidenced by the high bilateral foreign direct investment (FDI) between the two nations. A plot of the FDI between the U.S. and China

is illustrated in Figure 1 by CaixaBank Research [9], emphasizing the FDI for technological goods. We will be describing the series of events that cause the massive changes in FDI beginning in 2018.

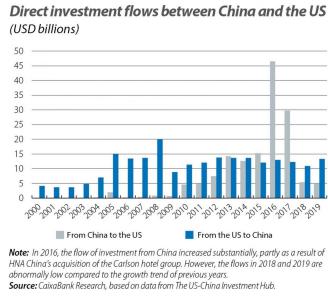


Figure 1: FDI between U.S. and China

Secondly, let us examine the current state of the relevant technology industries. First and foremost is the semiconductor industries, which can be considered one of the most (if not the most) influential industries in this world where having the advanced digital technology means having an edge in innovation and profit in general (as well as national security). We may roughly divide this into two subindustries: the foundry industry (who manufactures the silicon chips, and advances the process node technology), and the chip vendor industry (who designs and sells the chips manufactured by the foundries).

In the foundry industry, there is a clear oligopoly by TSMC and Samsung, who have 55% and 17% of the global foundry market share by Q3 2021, respectively [13]. Other major players include Global Foundries, which has a 6% market share [10]. Intel also has its own foundry services, but it only manufactures its own chips, up until the recent creation of the Intel Foundry Services in Q1 2022 [14]. This monopoly is what the textbook would classify as an ownership monopoly, since these large and established companies are the only ones with the access to the technology needed to manufacture advanced chips. Notably, these foundries are very long-term investments and cannot be quickly erected; or, as Bajarin from Forbes notes, "semiconductor foundries are not startup opportunities" [12]. In the chip vendor industry, there are more players, and it is less of an oligopoly overall. However, once you look at any specialized target audience, there are clear oligopolies. For example, in the consumer desktop/laptop CPU market, there are AMD, Intel, and Apple [16]; in the smartphone CPU market, there are Samsung, MediaTek, and Qualcomm [15]; in the GPU market, there are AMD, Nvidia, and Intel [17]; in the server CPU market there are AMD, Intel, and ARM CPUs [18]; etc. for server and ASICs/FPGAs markets. These markets are probably mostly legal and ownership oligopolies, due to legal patents and ownership of human capital and entrepreneurship.

In the communications industry, there is a global oligopoly as well. Huawei is the leading company in telecommunications equipment sales, accounting for 28.7% of global market share in Q3 2021, according to the Dell'oro Group; the top seven companies account for 80% of the global market share [19]. The China Academy of Information and Communications Technology (CAICT) showed that Huawei ranked first with 35.2% market share in the first half of 2021 in 5G equipment [6].

Another industry important to this discussion is the sale of consumer electronic devices such as smartphones. One of the notable players is Huawei, whose name we just encountered in the communications industry. Huawei led smartphone sales in 2020 (overtaking Samsung as first place, and pushing Apple to third place) in 2018, according to a report by Canalys in July 2020 [3]. In fact, Huawei is a highly diversified Chinese company who leads global sales in multiple industries, such as inverters (the electronic component) and having the fastest-growing mobile OS, HarmonyOS [6]. Huawei will be the most important company to play a part in the U.S. export sanctions, in part leading to its great diversification.

We should note that in this discussion of oligopolies that its relationship to inequality is not widely considered here. Typically, discussions of monopoly or oligopoly are intimately tied to a decrease in efficiency and an increase in inequality. Most of these advanced technological markets are dominated by some of the largest companies in the world, and are thus monopolistic markets rather than competitive markets (using the textbook's terms). This means that there is a high barrier of entry to these markets, and the set of companies that form the monopoly are relatively stable. The only likely realistic way for new companies to enter these markets is through massive government subsidies, which is a technique we will see is employed often by the Chinese government.

We cannot totally study the technology industries in isolation, because they tend to have many externalities. There is a generally positive externality of technological advancement, in that it improves efficiency in general and gives people better-performing devices. But there are also many more complicated issues, especially in the context of breaking copyright laws and stealing designs, which further advance the technology for the thief, but disadvantage the victim of the thievery. In addition, since the technological industries tend to be very oligopolistic, there is the negative externality of promoting inequality nationally or globally. Each of these externalities may have many additional effects. For the sake of this paper, we are only concerned with the primary effects of the technological trade between the U.S. and China and the immediate effects of this trade.

Causes of the U.S. export sanctions

In an ideal and simple world, societies would benefit most by producing the goods and services for which they have a comparative advantage, and trading those goods and services with other nations for goods and services for which the nation does not have a comparative advantage. However, of course the world is not so simple, as the world is always innovating and comparative advantage is always slightly shifting.

A country may attempt to bolster its ability to produce a particular good or service so that it may have a greater comparative advantage. One way to do so is to invest in the industry, either by subsidizing local production, or by acquiring foreign companies. Another way is to "cheat" by unlawfully stealing designs from other nations.

We see both of these in the influential case of Micron in recent years. Micron is a major chip designer and foundry in the U.S. In 2015, the Chinese state-owned company Tsinghua Unigroup attempted to

acquire Micron for \$23 billion, a move that was stopped by the U.S. government [8]. This is part of a 2014 Chinese initiative called the National Integrated Circuitry Investment Fund aimed to build a native memory chip manufacturing sector [21]. This move was not approved by the Committee on Foreign Investment in the United States (CFIUS) due to concerns about national security. After the failed takeover, a Chinese competitor, Fujian Jinhua, allegedly stole innovations from Micron. After allegations from Micron about the theft, China blocked Micron from exporting to China [8]. This shockingly dishonest business practice was one of the strongest pieces of evidence for Trump's export bans.

Huawei also recently became big news in 2019 when it was placed on the Entity List (a list of companies that may only be traded with with explicit approval from the U.S. government) due to claims of intellectual property theft, which prompted the beginning of the most severe export sanctions that still exist now, three years later [1]. However, the U.S. government has been wary of Huawei for many years in its communications industry. In 2008, Huawei's acquisition of the United States company 3Com Corporation failed to pass CFIUS review, also for national security concerns [6].

In addition to individual companies attempting to acquire or steal information from American companies, there are also large campaigns by the Chinese government to heavily subsidize the technological industries. The National Integrated Circuitry Investment Fund was related to the attempted Micron takeover. Another major initiative is the Made in China 2025 plan, in which the Chinese government has committed \$120 billion towards domestic semiconductor manufacturing, with the goal of domestically producing 70% of the chips for local consumption [1]. There is also the "Proposal of the Central Committee of the Chinese Communist Party on Drawing Up the 14th Five-Year Plan for National Economic and Social Development and Long-Range Objectives for 2025," a Chinese initiative that puts strengthening science and technology at the forefront for the purposes of national security [6]. Additionally, China requires that foreign firms form joint ventures with domestic Chinese firms as a condition for market entry in some economic sectors, which many consider as a form of "forced technology transfer" [4]. In summary, hoarding of the most advanced electronics technology is at the heart of national security talks for both the U.S. and China.

[U.S. export sanctions and aftermath](#)

In March 2018, the United States Trade Representative (USTR) released findings from its Section 301 investigation, which found China's acts and policies related to tech trade "unreasonable and discriminatory and burden or restrict U.S. commerce" [5]. They mention the joint venture requirements and "cyber intrusions into U.S. commercial computer networks," strategic acquisition of American companies, and strategic subsidizing as methods that "deprive the U.S. of the ability to set market-based terms in licensing and other tech negotiations" [5]. As a result, the USTR released two lists of Chinese imports totaling \$50 billion to be taxed with a 10% tariff. The USTR decided that these did not have the intended impact, and thus increased the tariff to a list of \$200 billion worth of Chinese imports, to be taxed at a rate of 25% beginning in 2019. This tariff is not restricted to technological goods, but is rather part of the more general U.S.-China "trade war" of the Trump era.

Sanctions on Huawei also began to increase greatly along this time, with AT&T and Google breaking contracts with the company in 2018 [6]. In May 2019, Huawei was placed on the Entity List, along with 114 of its affiliates [1]. This prevents the sale of U.S. products to the company without explicit approval from the government. An amended export rule prevented shipments to Huawei from any company (including non-U.S.-based companies) that deals with Huawei and with the U.S. without explicit U.S. government approval [1]. This greatly further challenged the company, as it could no

longer do business with any U.S.-dealing company, such as TSMC. TSMC previously manufactured Huawei's custom-designed high-end chips, called HiSilicon, and China-only foundries are much less advanced [2]. The sanctions against Huawei were strengthened again in June 2021 when President Biden passed the United States Innovation and Competition Act of 2021 (USICA), which prevents the removal of Huawei from the Entity List without demonstrating that it no longer proves a threat [6].

Clearly, these tariffs and trade restrictions are intended to be punitive towards China, aiming to coerce China into falling in line with U.S. trade and copyright restrictions. Companies such as Micron initially celebrate trade restrictions, but was quickly hit by the export restriction to Huawei, since Huawei represents 13% of Micron's sales at the time of the export rule [8]. This is true of any domestic technology manufacturer in the U.S., and it seems that all experts agree on one thing (as well as the teachings from our course): these export restrictions hurt everyone in the long run, and not least U.S. producers. The United States SEMI industry group protested against President Trump's proposal to impose an export ban on China's Semiconductor Manufacturing International Corporation (SMIC) [1]. Similarly, suppliers have lobbied against the passing of the Export Control Reform Act (ECRA), and a large number of exceptions to the export rules have been passed on a case-by-case basis to help suppliers [6]. The president of the Information Technology and Innovation Foundation (ITIF), a U.S. think tank, summarizes: "Let's be clear, the trade war has been very bad for the semiconductor industry in many ways" [8]. Using our microeconomics knowledge, we know that any sort of trade restriction or tariff imposes a deadweight loss that decreases the size of the economy. Additionally, export restrictions decrease demand and hurt domestic suppliers' revenue.

The next question is whether the punitive actions have the intended effects. The answer here is also no. If we consider overall trade between U.S. and China, we see that trade and foreign investments have both rebounded since 2018 [4]. Another major indicator is that Huawei's current business is booming, as it remains the top communications equipment vendor and one of the top smartphone vendors. For example, by the end of 2021, Huawei had signed more than three thousand commercial contracts for 5G industry applications, and HarmonyOS is the fastest growing mobile operating system after Huawei was banned from using Android on its devices [6]. This is despite the strengthened sanctions on Huawei, and partly due to diversification to other industries such as self-driving cars and green energy.

Present day, normative recommendations, and conclusions

It is clear that the trade war did not have the intended punitive benefits, and caused a lot of collateral damage to suppliers in the U.S. In particular, the trade restrictions do nothing to curtail China's use of subsidies, which play a large part in China's strategy [8]. The Carnegie Endowment states that "punitive trade measures have had little effect on altering economic outcomes, and sanctions usually do little to cause governments to change their core beliefs" [4], and this is clear with China hardly changing their practices throughout. There has been some improvement in the enforcement of intellectual property rights, however: a 2020 Business Climate Survey by the American Chamber of Commerce in China found that 70% of surveyed U.S. firms in China felt that enforcement of intellectual property rights in China have improved [4]. Carnegie Endowment also states that this improvement may simply be a matter of time, since it takes generations for sound intellectual property rights to emerge, as it did in the U.S.

Luckily, the hostility that pervaded the Trump presidency is fading. Andy Purdy, Huawei's Chief Security Officer, is "heartened by the company's ability to shift business and grow profits," and states that there's been a "calming down of the rhetoric that was used a couple of years ago" [7]. The sanctions that blanketed many categories of goods from the initial USTR tariffs in 2018 have now

become more fine-grained and strategic, “to address the comprehensive challenge of China’s rise to the U.S. in the political, economic, military, scientific and technological, diplomatic, and humanistic fields” [6], which prevents too many U.S. suppliers from being needlessly affected.

One of the important lessons learned, particularly from Huawei, is the importance of diversification. Despite being given seemingly-insurmountable obstacles that essentially shut its supply chain off from the U.S. and the U.S.’s trade partners, it managed to maintain the global leader in communications equipment and even maintain competence in its smartphone sector. It shows a resiliency and innovation that any technology company should strive for.

To truly tackle the issue of dishonest business practices and intellectual property theft, many people have proposed the idea of international alliances or committees aimed at tackling these issues. For example, several leaders of important tech firms in the U.S. have formed the China Strategy Group (CSG) and have proposed the creation of the “T-12,” an alliance of twelve nations dedicated to tackling such issues [11]. In particular, this would advocate and monitor a “disentangling” of the tech spheres of the U.S. and China – the alternative to which is “a world in which China’s non-democratic norms have ‘won.’” Another alliance called “Five Eyes” dedicated to monitoring China in the view of international public opinion on human rights and cybersecurity has been proposed [6].

One thing that all nations have to be worried about is the possibility of a “splinternet,” which will be a huge negative impact for all nations due to the immense impact of the Internet. The Internet is more or less a global public good, since it is non-exclusive and non-rival (assuming that the infrastructure exists); making certain areas exclusive will greatly hamper its ability. If the T-12 calls for a global bifurcation of our technological markets, it must not be an absolute bifurcation to avoid the risk of such a dramatic split. More generally, this should emphasize that we need to create a global arena of respect; the existence of global alliances should never alienate a country beyond reason, otherwise nations will revert to similar punitive methods that have no effect out of a populist antagonism.

Finally, the most straightforward recommendation is to simply reciprocate China’s strategic subsidizing of the advanced technology sector. This seems to be the most successful strategy, and is not inherently devious. China has laid out very clear plans with its Made in China 2025 campaign and similar plans, allowing it to rise to be a global superpower in technology in a very short period of time. This plan does not tackle the problem of intellectual property theft or other unethical practices; however, I personally believe that these are unavoidable to some degree. Instead of trying to constantly punish the other side (and often to no avail), focus instead on innovating at a fervid pace, and always stay one step ahead of the imitations. In other words: try to improve the U.S. domestic companies, without worrying about punishing China.

While it is idealistic to completely ignore China, I believe that this method is beneficial because it is simple, non-antagonistic, does not incur a dead-weight loss by imposing trade barriers, and focuses all efforts on maximum innovation (rather than diverting some of that effort and funding to punitive actions). Luckily, there are a number of large spending initiatives in the U.S. similar to China’s campaigns dedicated towards strategically keeping U.S. technologies at the forefront. These include the Creating Helpful incentives to Produce Semiconductors (CHIPS) Act, the American Foundries Act [1], and the America Creating Opportunities for Manufacturing Pre-Eminence in Technology and Economic Strength (American COMPETES) Act of 2022 [6]. These each allocate billions of funding towards development in semiconductor and related industries.

In summary, I believe that a strong push towards subsidizing domestic innovation, mixed with special interest groups or targeted international alliances to act as advisory committees, is a good general framework to tackle the current U.S.-China technological trade war.

References

- [1] Ioannou, Lori. "A Brewing U.S.-China Tech Cold War Rattles the Semiconductor Industry." *CNBC*, CNBC, 19 Sept. 2020, <https://www.cnbc.com/2020/09/18/a-brewing-us-china-tech-cold-war-rattles-the-semiconductor-industry.html>.
- [2] Kharpal, Arjun. "'Bleak but Salvageable': Huawei Has Limited Options as U.S. Sanctions Cut off Supply to Smartphone Chips." *CNBC*, CNBC, 12 Aug. 2020, <https://www.cnbc.com/2020/08/12/huawei-options-as-us-sanctions-cut-its-supply-of-smartphone-chips.html>.
- [3] Kharpal, Arjun. "Huawei Overtakes Samsung to Be No. 1 Smartphone Player in the World Thanks to China as Overseas Sales Drop." *CNBC*, CNBC, 30 July 2020, <https://www.cnbc.com/2020/07/30/huawei-overtakes-samsung-to-be-no-1-smartphone-maker-thanks-to-china.html>.
- [4] Huang, Yukon. "The U.S.-China Trade War Has Become a Cold War." *Carnegie Endowment for International Peace*, Carnegie Endowment for International Peace, 16 Sept. 2020, <https://carnegieendowment.org/2021/09/16/u.s.-china-trade-war-has-become-cold-war-pub-85352>.
- [5] "USTR Finalizes Tariffs on \$200 Billion of Chinese Imports in Response to China's Unfair Trade Practices." *United States Trade Representative*, United States Trade Representative, 16 Sept. 2020, <https://ustr.gov/about-us/policy-offices/press-office/press-releases/2018/september/ustr-finalizes-tariffs-200>.
- [6] Chen, Dingding, and Wang Lei. "Where Is China-US Technology Competition Going?" – *The Diplomat*, For The Diplomat, 5 May 2022, <https://thediplomat.com/2022/05/where-is-china-us-technology-competition-going/>.
- [7] Fried, Ina. "The Chinese Tech Giant Trump Couldn't Kill." *Axios*, 29 Mar. 2022, <https://wwwaxios.com/2022/03/29/huawei-chinese-tech-giant-trump-couldnt-kill>.
- [8] Swanson, Ana, and Cecilia Kang. "Trump's China Deal Creates Collateral Damage for Tech Firms." *The New York Times*, The New York Times, 20 Jan. 2020, <https://www.nytimes.com/2020/01/20/business/economy/trump-us-china-deal-micron-trade-war.html>.
- [9] Canals, Clàudia, and Jordi Singla. "The US-China Technology Conflict: An Initial Insight." *CaixaBank Research*, 9 Nov. 2020, <https://www.caixabankresearch.com/en/economics-markets/activity-growth/us-china-technology-conflict-initial-insight>.
- [10] Alsop, Thomas. "Top Semiconductor Foundries Market Share 2021." *Statista*, 12 Apr. 2022, <https://www.statista.com/statistics/867223/worldwide-semiconductor-foundries-by-market-share/>.

- [11] Allen-Ebrahimian, Bethany. "Former Google CEO and Others Call for U.S.-China Tech 'Bifurcation.'" Axios, 26 Jan. 2021, <https://wwwaxios.com/2021/01/26/scoop-former-google-ceo-and-others-call-for-us-china-tech-bifurcation>.
- [12] Bajarin, Tim. "The Semiconductor Industry's Competitive Dilemma." *Forbes*, Forbes Magazine, 4 Mar. 2021, <https://www.forbes.com/sites/timbajarin/2021/03/04/the-semiconductor-industryscompetitive-dilemma/>.
- [13] Park, Shin-Young. "Samsung Overtakes Intel as Foundry Looms as next Battlefield." *KED Global*, KED Global, 2 Aug. 2021, <https://www.kedglobal.com/semiconductors/newsView/ked202108020009>.
- [14] Its Fabless Chips Rivals Flat-Footed." *Fortune*, Fortune, 15 Feb. 2022, <https://fortune.com/2022/02/15/intel-tower-semiconductor-foundry-acquisition-fabless-chips-rivalsnvidia-amd-qualcomm-tsmc/>.
- [15] Weissberger, Alan. "Counterpoint Research: Mediatek Is World's #1 Smartphone Chipset Vendor." *Technology Blog*, 25 Dec. 2020, <https://techblog.comsoc.org/2020/12/25/counterpoint-researchmediatek-is-worlds-1-smartphone-chipset-vendor/>.
- [16] Hruska, Joel. "Apple, AMD, and Intel Are Pursuing Three Different Strategies to Win the Laptop Market." *ExtremeTech*, 25 Feb. 2022, <https://www.extremetech.com/extreme/330417-apple-amd-intel-are-pursuing-three-different-strategies-to-win-the-laptop-market>.
- [17] "[Q3 2021 Overall GPU Market Share]." *Hardware Times*, 16 Dec. 2021, <https://www.hardwaretimes.com/amds-dgpu-market-share-4-1-nvidia-20-intels-igpu-share-at-62-q3-2021-overall-gpu-market-share/>.
- [18] Alcorn, Paul. "AMD Sets All-Time CPU Market Share Record as Intel Gains in Desktop and Notebook PCs." *Tom's Hardware*, Tom's Hardware, 9 Feb. 2022, <https://www.tomshardware.com/news/intel-amd-4q-2021-2022-market-share-desktop-notebook-serversx86>.
- [19] Fletcher, Bevin. "Huawei Still Dominates Telecom Equipment Market." *Fierce Wireless*, 16 Dec. 2021, <https://www.fiercewireless.com/wireless/huawei-still-dominates-telecom-equipment-market>.
- [20] Thomas, Will. "US Research Security Campaign under Strain as Cases Falter." *American Institute of Physics*, American Institute of Physics, 25 Jan. 2022, <https://www.aip.org/fyi/2022/us-research-security-campaign-under-strain-cases-falter>.
- [21] Shields, Anne. "Micron Gets a \$23 Billion Takeover Offer from Tsinghua." *Yahoo!*, Yahoo!, 22 July 2015, <https://www.yahoo.com/entertainment/s/micron-gets-23-billion-takeover-130924921.html>.

TeachEcoKnowmics Blog Post 1

Jonathan Lam

2022/01/30

This week I read "Former Google CEO and others call for U.S.-China tech 'bifurcation'" by Bethany Allen-Ebrahimian on Axios, written almost exactly a year ago. This article concerns the report by the "China Strategy Group" formed of a number of high-ranking executives at technology companies in the U.S., such as Google's CEO and several of its employees. In this report, they state that they are concerned that America's leadership in technology is at risk due to China's difference in values, in particular their willingness to disregard digital licensing laws. In class we discussed how important innovations in the technology sphere are for developed nations in terms of percentage of overall production and income, so this is a dire scenario. And the sale of goods or services (in this case, technological services) is greatly hindered if one party doesn't regard the other's wishes or values; this leads to the selling party ("unfairly") losing income or expending effort in order to protect said lost income, and a general discomfort between the parties to participate in future relationships. The CSG suggests several proposals for the U.S. leadership: a "bifurcation" or separation of some of our technological markets from China's ones; redesigning government and shifting governmental expenditure towards "tech analysis and forecasting"; improving domestic infrastructure, "ally-centric production," and an "alliance of democracies called the 'T-12'" that decreases our reliance and connection with Chinese technological markets. To me, this sounds very much like former President Trump was doing with anti-competition laws for China. This topic is interesting because anti-competition will maintain the incentive for American companies to innovate and remain at the forefront of the tech world; however, anti-competition in general hinders progress, such as cheaper ways to produce the same leading technologies at similar or slightly-lower companies (which may have legitimate uses in lowering costs for the people who do not need the best products from the most innovating companies). As someone in the tech world and heavily interested in Google, I will watch the trajectory of this subject with great interest.

TeachEcoKnowmics Blog Post 2

Jonathan Lam

2022/02/07

I read the MarketWatch article "Billionaire investor Ray Dalio on capitalism's crisis: The world is going to change 'in shocking ways' in the next five years" by Jonathan Bittner, published September 2020. This article comprises an abridged interview with Ray Dalio, billionaire and founder of the world's largest hedge fund. As the title suggests, Dalio describes the current state of capitalism in the U.S., and particularly its issues with inequality of opportunity. He suggests that capitalism is very good at increasing the size of the "economic pie," but not very good at dividing it up amongst the people (while socialism is not as good at increasing the size but better at dividing it). This forms one of the three big issues that will continue to plague the U.S. for the next 5-10 years: wealth and values gap, money and credit cycles, and the emergence of other dominant world financial powers like China. Dalio argues that the combination of these problems, as well as the loss of an education advantage, puts the U.S. in the same situation as the Dutch and British empires shortly before falling.

A year and a half have passed since this article was written, and yet many of his words still ring very true. At the time of writing, the latest presidential election had not occurred yet, and the coronavirus was in its first year of rampage: a time of great uncertainty. Now, we've reached almost a predictable equilibrium with the coronavirus in terms of work-life balance, but the increase in inequality that it caused is clear to be seen. We've seen technology companies and billionaires grow enormously in wealth – just this last class, we noted that the top ten richest men doubled their wealth during the pandemic – while many lower-income people are still struggling to get back to work normally. There is the highly-relevant "Great Resignation" movement or "antework" movement (the latter named after a subreddit that recently took off) that protest working with the current wages and treatment by employers. Just as Dalio said, many people are resentful, and that hurts productivity by making many people not want to work. Also as Dalio said, inflation is getting much worse as a result; people need and want money to survive, and the rich are not willing to pay more taxes, so in order to get enough money to keep the public happy we need to print and borrow more money, and that is leading us into those debt and money-printing cycles that are very difficult to come out of. At this point, it's getting hard to see what is related to COVID-19 and what is going to continue to be a trend after health restrictions are lifted – at this point, I'm scared to predict what will happen in the next 5-10 years.

TeachEcoKnowmics Blog Post 3

Jonathan Lam

2022/02/20

This week, I read "How the Coronavirus Played Out by Sector and Demographics" by Dan Burns on Renton on April 14th, 2020. This article was written at a time when the shutdown of businesses was beginning and businesses were having massive layoffs. This is relevant to our recent discussions in class of supply and demand, since it clearly shows the complicated choices that businesses under pressure make about the labor market, and the relationship between the goods and the labor market. In the case of goods and services, the primary driver is demand because people are unwilling to go out and receive non-essential services. There is also a massive increase in the demand of certain items, such as hand sanitizer and masks at this time. In the labor market, we have a decrease in demand since fewer people want non-essential goods, but we also have a decrease in supply because service providers are also less willing to perform non-essential services. As we learned in class, this will generally decrease quantity of labor (higher unemployment, which agrees with the article) but price (wages) will stay roughly the same. In addition, the demographics and occupational fields determine the level of increase in unemployment: as the article notes, non-essential services such as restaurant workers and healthcare workers (except those working in COVID-19 suppression) are hit harder than others. Some industries even have an exceptional increase in demand, such as remote technology providers and scientists developing the COVID-19 vaccine.

At the time of the article, it would probably have been unfathomable to businesses that only two years later (at the present time), we would have the Great Resignation and so many people would be voluntarily quitting at a time when unemployment is already high due to the aforementioned factors.

TeachEcoKnowmics Blog Post 4

Jonathan Lam

2022/03/08

This week I read "These countries may soon have the highest life expectancies," by Ashley Welsh on CBS News on October 18, 2018. Unlike the other articles I've read so far, this one was written before the coronavirus was widely known, and thus unencumbered by it. This is especially interesting as this article is about a health issue, and it would definitely be interesting to see what the current trends and predictions are and how they differ from just four years ago.

This article uses data and predictions from a medical journal called The Lancet. The study is simply about where countries currently stand with regard to their life expectancies, as well as their life expectancies in 2040 (22 years after the article was written). As expected, life expectancies are predicted to increase; however, on the lower-end of the predictions, life expectancies for almost half of the countries may decrease. In particular, the U.S., which has a very mediocre ranking of 43rd out of 195 countries, is expected to decrease to a ranking of 64th by 2040 with a modest life expectancy increase of only 1.1 years. Japan, which currently holds the top spot, is expected to become second to Spain in 2040.

The article explains that there is a range of predictions, due to whether we address "key risk factors, levels of education, and per-capita income." This is stated before the extended coronavirus pandemic is known, which definitely affects life expectancies. Not only does the coronavirus directly cause an increase in deaths (key risk factors?), it is also widening the inequality gap (levels of education and per-capita income). And now, with the latest war between Russia and Ukraine, financial anxiety is still very high with the high rates of inflation and the international conflict in Eastern Europe (which may also affect life expectancies in many ways in the relevant countries, especially if the sanctions on Russia have long-term effects).

In summary, while medical journals may place trends on life expectancies based on medical data and other trends, it seems that there is a large volatility in the world at this time (health-wise, economically, and politically) that may cause large variations in future health and life-expectancies.

TeachEcoKnowmics Blog Post 5

Jonathan Lam

2022/04/12

This week, I watched the video “AI’s impact will extend to 100% of jobs” posted by Bloomberg, originally published November 1st, 2019. This is an interview of IBM’s Vice President of People and Culture, Obed Louissaint. As the title suggests, Louissaint believes that AI will affect all jobs in the near future. He starts by looking at the changes in valuation of tasks over the near future: repeatable tasks will lose value as they are overtaken with automation, whereas soft skills will gain value over time because of their difficulty in being automated. In manufacturing, this is due to a movement of jobs from middle-class jobs to both lower- and higher-class jobs. Louissaint says that IBM’s role in this shift is to identify skills that are valuable and target these skills. They developed an AI tool called YourLearning that actively tracks 15,000 workers daily in order to give them the skillset that would be helpful in this economy.

I find it interesting how we talk about how technology replaces repetitive jobs, thus hurting the people whose jobs were replaced. On the other hand, we also have technology (YourLearning) that is helping people to switch their competencies to those which are higher valued. This is an interesting case of supply-and-demand: we have a surplus of people and machines who can do repetitive tasks, and a relative shortage for people who can do jobs with more interpersonal skills. Normally, people would naturally adjust their skillsets so that they can have higher-paying jobs where the demand is highest, and thus make a more efficient labor market. However, since automation is changing the labor market so quickly by replacing jobs, we find the use of tools such as YourLearning to help facilitate this efficient equilibrium very useful to workers.