COI: A First Step towards TrueType Bytecode Analysis

by

Wenzhu Man

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

Abstract

TrueType fonts are ubiquitous in today's computing environment. While TrueType is a scalable vector-based font format, optimal scaling requires font hinting for excellent font rendering at typical screen resolutions. TrueType font designers express font hints in a TrueType-specific stack-based bytecode language. Common font manipulations, such as font subsetting and merging, would therefore profit from the ability to analyze TrueType bytecode. Current tools must either omit bytecode, sacrificing readability, or conservatively generate duplicate bytecode, thus creating larger font files with considerable unused bytecode.

Our work takes a first step towards TrueType bytecode analysis. We describe the design and implementation of COI, a new three-address intermediate representation for TrueType bytecode. COI greatly simplifies optimization for TrueType bytecode. We have implemented a tool that translates between TrueType bytecode and COI and performs standard static analyses on COI. Our experiments show that COI can enable the removal of unused bytecode, thus reducing font size.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Patrick Lam, for his consistent and invaluable mentorship, without which, this thesis would not have been possible.

I would also like to acknowledge funding from Canada's Natural Science and Engineering Research Council, as well as the assistance of Behdad Esfahbod at Google, who posed the problem of TrueType font analysis and helped guide the research.

In addition, I would like to thank the readers, Professor Derek Rayside and Professor Gregor Richards, for their valuable time and comments.

Finally, I would like to thank my colleagues Jonathan Eyolfson and Xavier Noumbissi for helping me improve my work.

Dedication

I dedicate this thesis to my precious family and friends. I would love to thank my parents, Qilun Man and Hairong Zhao for their unconditional love and support throughout all my years. I also want to thank all my friends that help me during my study in this university.

Table of Contents

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

1.1 Approach 2

1.2 Contribution of Thesis 2

1.2.1 Design 3

1.2.2 Implementation 3

1.3 Organization 3

2 Related Work **4**

2.1 Automatic Font Hinting 4

2.2 TrueType Bytecode Manipulation and Validation Tools 5

2.3 Stack-based Bytecode Translation 5

2.4 Abstract Interpretation and Symbolic Execution 6

3 Background **7**

3.1 Computer Font 7

3.2 TrueType Fonts 8

3.2.1 TrueType Bytecode 8

3.2.2 Glyph Program, Font Program and the Control Value Program . . 8
3.2.3 The graphics state 8
3.2.4 The storage area and CVT table 9
3.2.5 Information from the execution environment 9
3.3 Font Engines 9
3.4 Font Subsetting 10

4 Motivation **11**
4.1 Why do we need TrueType bytecode analysis and optimization? 11
4.2 Why we can't transform bytecode directly? 11
4.2.1 Advantages of stack-based languages 12
4.2.2 Disadvantages of optimizing TrueType bytecode directly 12

5 Design and Implementation **17**
5.1 Abstract Interpreter 17
5.1.1 Observations 17
5.1.2 Type Inference 19
5.1.3 Accurate Stack Height 20
5.1.4 Implementation 20
5.2 Translating TrueType to COI 23
5.2.1 Properties of COI 23
5.2.2 Assignment Statements 24
5.2.3 Arithmetic and Logic Operations 24
5.2.4 IF/ELSE/EIF 26
5.2.5 Function call 27
5.2.6 Reads and writes from storage area and CVT table 28
5.2.7 Instructions that manage the stack 28
5.2.8 Graphics State Related Instructions 30

5.2.9 Other 31
5.3 Possible Optimizations 31
5.3.1 Constant propagation 31
5.3.2 Dead code elimination 31
5.3.3 Branch Elimination 32
5.4 Experimental Methodology 32

6 Experimental Results **33**

7 Conclusions and Future Work **36**
7.1 Discussion 36
7.2 Limitations 36
7.3 Future Work 37

A Experimental results in detail **39**

References **50**

List of Figures

4.1 TrueType bytecode computing (PPEM >2047) OR (PPEM <9) 13

4.2 Function labels are implicit in TrueType 15

4.3 CALL instructions' destinations are implicit in TrueType bytecode 16

5.1 Relationships between COI components. Dashed box not yet implemented. 18

5.2 Implementation of ROUND in abstract interpreter 20

5.3 TrueType bytecode code which includes IF-ELSE instructions 21

5.4 Example of Function Definitions After Preprocessing 22

List of Tables

5.1 COI Representations of TrueType Arithmetic and Logic Operations 25

6.1 Counts of reachable functions in selected Google Noto Sans fonts. Under 15% of bytecode functions are reachable. 34

6.2 Counts of static callers to functions are concentrated for the KufiArabic, SansBoldItalic, SansBold, and SansItalic fonts. 35

A.1 Functions Called from Selected Glyph Programs of NotoKufiArabic 39

A.2 Functions Called from Selected Glyph Programs of NotoSansBold 42

Chapter 1

Introduction

TrueType is an outline font standard designed by Apple and Microsoft. It was first introduced in the Macintosh System 7 operating system in 1990 as a competitor to Adobe's Type 1 fonts used in PostScript. It has now become the most common format for fonts on both the Mac OS and Microsoft Windows operating systems.

TrueType font files contain a set of glyphs (or characters), where each glyph is described by a set of outlines. The outline of each glyph is usually represented by a set of splines. Sometimes, professional typographers or auto-hinting tools insert instructions into TrueType fonts to adjust the display of an outline font and better preserve the original outline of the character at low screen resolutions. These executable instructions are called TrueType bytecode.

Fonts without hinting use a naive outline-filling approach to render glyphs onto a grid of pixels: each pixel whose center lies inside an outline is set to black (or other colors) after the outlines are scaled to the desired size.

Unlike fonts without hinting, which simply rely on vector outlines, hinted fonts additionally use TrueType bytecodes to adjust the font outline and control glyph display. TrueType bytecode instructions represent a sequence of grid-fitting rules which ensure that characters line up well, so that the text rendered in that font looks as smooth and legible as possible, even when scaled to low resolutions.

As TrueType bytecode is an important part of TrueType fonts, we need to manipulate TrueType bytecode when manipulating font files. Examples of manipulation include font merging and font subsetting. Font subsetting is a technique for reducing font file size by only including characters actually used in a document. However, due to lack of TrueType bytecode optimization techniques, current subsetting techniques does not touch the bytecode from the full font file. Instead, current tools copy all bytecode from the full font file to the subset, which may produce a considerable volume of unused bytecode in subset font files and contribute to the unnecessary usage of system resources such as the network.

As the popularity of smart phones and tablets continues to increase, so does the popularity of TrueType fonts on these smaller hand-held devices. Due to physical storage and memory limitations on these devices and the high Internet latency on mobile networks, reducing the space used by fonts would be helpful. To achieve this, the demand for TrueType optimization techniques becomes more and more urgent.

1.1 Approach

TrueType bytecodes are stack-based. This stack-based model complicates transformations, as statements and expressions are not explicit in this model. In contrast, three address code is widely used as an intermediate language in modern optimizing compilers to enable optimizing transformations.

We chose to design a new three address code to mitigate the complexities of dealing with TrueType's stack-based bytecode.

This thesis therefore proposes COI, a new three-address code designed specifically for TrueType bytecode.

1.2 Contribution of Thesis

This thesis takes the first step towards TrueType bytecode analysis and optimization. Current TrueType bytecode is produced by professional typographers and interpreted by font engines. Transformations and optimizations of bytecodes in existing fonts are challenging.

The contributions of this thesis are the design and implementation of COI, the first three-address code intermediate representation for TrueType bytecode. COI was designed to simplify the process of optimizing TrueType bytecode.

We have implemented a tool that transforms TrueType bytecode to COI, in the context of the popular fonttools open-source project for font manipulation [1]. Our tool incorporates an abstract interpreter for TrueType bytecode as part of the transformation workflow.

1.2.1 Design

The design of this thesis can be split into two parts: first, an abstract interpreter of TrueType bytecode; second, the three-address code COI. COI was designed to simplify the process of optimizing TrueType bytecode.

1.2.2 Implementation

We implemented a tool that transforms from TrueType bytecode to COI. Our plan is to integrate this tool into the fonttools project.

1.3 Organization

The remainder of the thesis is organized as follows. In Chapter 2, we introduce the related work of this thesis.

In Chapter 3, we introduce background for this thesis: Digitizing Letterform Designs, the nature of TrueType bytecode, current font engines, and current font subsetting and merging techniques.

In Chapter 4, we discuss the motivation for this thesis: the lack of TrueType bytecode optimization techniques and why we need TrueType bytecode optimization techniques. We will also list the features of TrueType bytecode and how they affect the implementation of current font engines and current font subsetting and merging techniques.

In Chapter 5, we present the design and implementation of this thesis: an abstract executor and a three-address code which will greatly simplifies the TrueType bytecode optimization. In addition, we will explain how our tool will greatly simplify the TrueType bytecode optimization.

In Chapter 6, we present experimental results for this thesis.

Chapter 2

Related Work

The related work of this thesis can be divided into four categories: automatic hinting; bytecode manipulation tools; and compiler-related work on handling stack-based code and on symbolic execution.

2.1 Automatic Font Hinting

TrueType hinting is a time-consuming painstaking process, and numerous others have worked on automatic font hinting. We describe three automatic font hinting techniques.

Shamir presented a constraint-based approach for automatic hinting [16]. His method is capable of identifying hints inside characters, gathering global font information, and linking the font information to characters. The collected information is then organized into font hints used in many widely used hinting technologies, including TrueType and PostScript. This automatic font hinting approach can provide a solid basis for further font optimization and its generic nature can greatly help font designers.

Zongker et al proposed an approach which automatically hints fonts using examples at typical display sizes and screen resolutions [20]. Their automatic hinting method matches the outlines of corresponding glyphs in all fonts, and then translates existing hints for glyphs from the source to their specific target fonts. Their methods can help typographers generate hinting for newly created fonts: typographers can simply carry over existing custom modifications from existing fonts.

Visual TrueType [17] is a tool, developed by Stamm, which helps font designers produce TrueType fonts without resorting to low-level assembly code. It represents the design of TrueType fonts with a specialized glyph data structure which can be visualized and authored graphically. This structure is then automatically inserted into TrueType assembly code. Visual TrueType helps designers develop new fonts without having to understand the arcane details of TrueType ByteCode structure.

Automatic hinting tools greatly simplify the work for font designers. However, such tools cannot guarantee the quality of the generated hint code. The hint code could be very inefficient. Our tool could provide optimizations for the generated TrueType code such as dead code removal. We anticipate our tool serving as the final step in automatic hinting tools to perform pre-release optimizations on generated TrueType bytecode.

2.2 TrueType Bytecode Manipulation and Validation Tools

FontTools/TTX is a tool for font file manipulation. It supports reading and writing of TrueType and OpenType fonts. It also converts TrueType and OpenType fonts (.ttf) to and from an XML-based file format (.ttx) [1]. The input of our tool is the XML-based files generated by FontTools. FontTools is also widely used as for font merging and subsetting. However, its font subsetting and merging functionalities do not manipulate the TrueType bytecode in font files. Our tool takes the first step towards the analysis of TrueType bytecode, which could provide valuable information for TrueType bytecode optimizers, as well as for font subsetting and merging.

2.3 Stack-based Bytecode Translation

AST-based intermediate code has been widely used in compilers and optimization frameworks. Simple C is an AST-structured intermediate language of C used in the McCAT compiler [8] project at McGill University. Simple C defines a clear structure for C programs, which facilitates the analysis and optimization for such programs. Influenced by SIMPLE, GIMPLE and GENERIC are both tree representations for C functions [14]. The difference between GIMPLE and GENERIC is that GIMPLE is a three-address representation and each statement is restricted to no more than 3 operands. GIMPLE and GENERIC serve as the middle end in the GNU Compiler Collection[1]. Also inspired by Simple, Jimple [19]

[1]http://en.wikipedia.org/wiki/GNU_Compiler_Collection

is a three-address intermediate representation designed to simplify analysis of Java bytecode. Jimple is used in Soot [18], a well-known framework for Java bytecode optimization. Jimple shows that a three-address intermediate representation could help solve difficulties with optimizing stack-based bytecode. The steps of our framework are heavily inspired by Soot. Soot achieves Java bytecode optimization in three steps: first, it translates Java bytecode into three different intermediate representation: BAF, JIMPLE, and GRIMP. BAF is a typed three address code. Compared to native Java bytecode, BAF is more suitable for transformation and optimization. Second, Soot applies several intra-procedural and whole program optimizations directly to the intermediate representations. Finally, Soot translates the optimized intermediate representations back into Java bytecode.

Similarly, as will be later shown in Figure 5.1, our design is: first, we transform TrueType bytecode into COI; then, we perform optimizations on COI. In the final step (not implemented yet), we would transform the optimized COI back into TrueType.

Soot greatly facilitates various developments for Java bytecode optimization. For example, Dexpler [5] leverages Soot to translate Android Dalvik bytecodes to Jimple, enabling future Android bytecode analysis and optimization.

TrueType is a less popular language than either C or Java, and there is no intermediate representation designed specially for TrueType bytecode. Inspired by the success of the intermediate representations listed above, we decided to design a three-address intermediate representation for TrueType.

2.4 Abstract Interpretation and Symbolic Execution

Abstract interpretation performs a partial execution of a program and gains semantic information about the program's data flow and control flow, without necessarily performing calculations on actual values. The main difference between abstract interpretation and normal (concrete) program interpretation is that values during the execution process may be abstract (rather than actual data). For example, values could be symbolic formulas over the input symbols. Symbolic execution of programs use symbols that represent arbitrary values; such symbols often occur when representing program inputs [12]. Symbolic execution is often used to explore every possible input of a program. Instead, we use abstract interpretation to get the information we need to perform TrueType transformations without doing unnecessary calculations and with unknown inputs.

Chapter 3

Background

In this section, we survey background material for this thesis, including digitizing letterform design, TrueType Fonts and TrueType bytecode.

3.1 Computer Font

A computer font is a data file that contains the shape information for a set of glyphs (also known as characters or symbols). Font engines render glyphs into matrices of dots or pixels. Each glyph can be rendered at many different sizes.

Generally, there are three broad categories of computer fonts: *bitmap fonts*, *outline fonts* (also called *vector fonts*), and *stroke fonts*. These categories differ in how they represent glyph shapes.

Bitmap fonts store the shape of each glyph using a matrix of pixels at each size. Rendering is straightforward. The major disadvantage of bitmap fonts is that they are non-scalable. Such fonts must be manually re-drawn at each size.

Outline fonts store shape information using Bézier curves[1]. They also include drawing instructions and mathematical formulæ to identify control points, which define the outlines for each glyph. Several outline font formats are: Type 1 and Type 3 fonts, TrueType fonts and OpenType fonts.

Stroke fonts define a glyph's outline using the vertices of individual strokes and stroke profiles. Stroke-based fonts are mainly used to describe East Asian character sets and

[1] http://en.wikipedia.org/wiki/B%C3%A9zier_curve

ideograms. For a font developer, editing a stroke-based glyph is less painstaking and less error-prone than editing outline-based glyphs. Stroke-based fonts are heavily used on embedded devices in East Asia due to their space saving properties.

3.2 TrueType Fonts

A TrueType font is described by a set of vectors (and the points contained therein). At the lowest level, each character (glyph) is described by a list of points on a grid [11]. Points are divided into two categories: off-curve points and on-curve points. On-curve points represent the end of a font outline while off-curve points act as control points. For instance, two on-curve points represent a straight line. If a third off-curve point is added between these two points, the three points together will represent a parabolic curve.

3.2.1 TrueType Bytecode

TrueType bytecode is a widely-used stack-based bytecode format for providing hints for fonts. bytecode instructions in font files define a sequence of grid-fitting rules to adjust the font outlines and control the display to produce clear, legible text under different screen resolutions.

3.2.2 Glyph Program, Font Program and the Control Value Program

Every glyph can provide a set of instructions in its *glyph program*. Instructions associated with an entire font as a whole constitute its *Font Program* (found in its TrueType fppm table) and *Control Value Program* (found in its prep table).

All the functions in a font file are defined in the *font program* and they can be called by all the *glyph programs*. The *font program* is executed only once. However, whenever the point size or transformation changes, the *control value program* will be executed by the font engine. The *glyph program* is executed when a specific glyph is requested.

3.2.3 The graphics state

The graphics state is part of the execution context of TrueType instructions. It includes a list of state variables. State variables are typed as booleans, integers, points or vectors.

The state variables are: auto flip, control value cut-in, delta base, delta shift, dual projection vector, freedom vector, instruct control, loop, minimum distance, projection vector, round state, rp0, rp1, rp2, scan control, scan control, single_width_cut_in, single_width_value, zp0, zp1, zp2.

All variables in graphics state have default values. Variables in the graphics state can be set and used by TrueType instructions.

Our tool only keeps track of state variables that influence the data on the program stack: loop, projection vector, and freedom vector.

3.2.4 The storage area and CVT table

The font engine maintains two portions of memory: the *storage area* and *CVT table*.

TrueType bytecode can exchange data between the storage area and the program stack using specific instructions.

The *CVT table* is defined and initialized in a font file's CVT table. TrueType instructions can read these values. The *CVT table* is used to store the value of certain font features.

3.2.5 Information from the execution environment

The instruction **MPPEM**[] puts the size in pixels-per-em for the current glyph onto program stack and the instruction **MPS**[] retrieves the size in points-per-em. The **GETINFO**[] instruction requests the font engine version number. This information is statically unknown and depends on the font engine's specific runtime execution environment.

3.3 Font Engines

TrueType font engines rasterize a glyph outline in three steps. First, the engine scales a master outline for the requested glyph. This step renders the font from device independent data to device dependent data, generating points at the appropriate size for the target device. In the second step, the engine executes the instructions embedded in the font data. These instructions move points to appropriate positions for the requested rendering resolution.

In the third step, the font engine's scan converter takes the font outline data and applies a set of rules to determine which pixels will be part of the glyph image the device is going to display. Finally, devices can display or print out the given font to end users.

3.4 Font Subsetting

An entire font may contain thousands of glyphs. However, rendering any document will only require a small fraction of those glyphs. For example, this thesis uses fewer than 100 glyphs. Embedding an entire font file into a document such as web page will cause inefficient space usage. Font Subsetting therefore makes a large font file into a smaller one that only includes characters that are actually used in the layout. For webpages whose selection of glyphs remain static after initial rendering, font subsetting allows content providers to minimize transmission sizes, which is very useful for mobile-compatible web pages.

Chapter 4

Motivation

We have previously described the TrueType font format, which includes TrueType bytecode. We now continue by discussing the motivation for TrueType bytecode analysis and optimization. We demonstrate the need for a new intermediate representation for TrueType bytecode by illustrating several difficulties which arise when attempting to optimize the stack-based TrueType bytecode directly.

4.1 Why do we need TrueType bytecode analysis and optimization?

Although each glyph in a font has its own bytecode (under the instructions tag), glyphs also share a global function table, defined under the fpgm tag. Due to the stack-based nature of TrueType bytecode, current subsetting techniques keep the entire global function table for every font file, producing a considerable amount of unused bytecode in font files.

As the purpose of font subsetting is to produce a font file which is as small as possible, the lack of TrueType bytecode analysis and optimization techniques harms the effectiveness of font subsetting techniques.

4.2 Why we can't transform bytecode directly?

To answer this question, we discuss the advantages and disadvantages of TrueType bytecode, which is stack-based.

4.2.1 Advantages of stack-based languages

In this section, we discuss the advantages of stack-based languages and speculate about why TrueType bytecode was originally designed as a stack-based language. Three advantages:

1. Instruction encoding is compact. As most operands are implicit, instructions can often be encoded as a single opcode without any operands. The instructions of stack-based virtual machines are called bytecodes mainly because opcodes are often a single byte long.

2. Interpreters for stack-based languages are more straightforward to implement. This property is very important for font files, which must be rendered on many different kinds of devices, some of which are heavily resource-constrained, and such devices require different font engines. The stack-based design of TrueType bytecode simplifies font engine development across different platforms. Moreover, the source code for stack-based language interpreters tends to be more compact, readable and more maintainable.

3. Syntax for stack-based languages tend to be less complicated. The specification of TrueType is quite simple compared to non-stack based languages: for instance, there are no variable names or declarations. All data is pushed onto the stack and the following operations execute directly on operands popped from the stack.

4.2.2 Disadvantages of optimizing TrueType bytecode directly

Here we list several critical reasons why we cannot optimize TrueType bytecode directly (i.e. without transformation).

1. Due to the complexities of stack-based languages, changing instruction order can be fairly complicated. Related instructions might exist arbitrarily far away from each other. For example, TrueType programs often push all data at the beginning of a program, followed by a sequence of statements. Therefore, there is no straightforward way to reason about which value is used by which statement. For this reason, identifying dependencies between instructions is quite challenging.

2. The complexities of TrueType bytecode increase with control flow—conditions for if statements and targets for function calls popped from the stack

3. Statements and expressions are not explicit. As all operands are popped from stack, there is no easy way to construct the expressions statically. In addition, expressions can be arbitrary large. For instance, Figure 4.1 shows code whose result evaluates to the expression v = (PPEM >2047) OR (PPEM <9).

Figure 4.1: TrueType bytecode computing (PPEM >2047) OR (PPEM <9)

```
MPPEM[ ]
PUSH[ ]
2047
GT[ ]
MPPEM[ ]
PUSH[ ]
9
LT[ ]
OR[ ]
```

4. Function labels are not explicit. The code snippet in Figure 4.2 defines two functions: Function 1 and Function 2. FDEF starts a function definition and ENDF ends a function definition. The instruction **FDEF** (function define) pops one element from the stack and treats that element as the function label. It is complicated to infer the value of the top of the stack, and we therefore say that the function label is implicit.

5. Callees are implicit and read off the stack. Furthermore, since the font engine has only one stack, a function will apply its actions to the font engine's single operand stack. As a result, when we encounter a sequence of function calls like the last three lines in Figure 4.3, there is no straightforward way to understand which three functions are called; the destination of the second CALL may in principle depend on the effects of the first function. There is no requirement that a callee leave the operand stack in the same state as upon entry (and indeed, function parameters and return values may be passed on the stack).

6. Reasoning about data flow is complicated. Some instructions, like **MPPEM** and **GETINFO**, will introduce implicit data flow. The instruction **MPPEM** depends on the graphics state. If the graphics state's projection vector is set to "x-axis", pushes the current number of pixels per em in x-direction onto the stack. Otherwise, it pushes the ppem in y-direction.

7. TrueType bytecode includes loop constructs. These instructions will read the loop state variable from the graphics state and execute the specified number of times. Furthermore, loop iterations are not required to leave the stack height invariant.

In conclusion, TrueType bytecode is designed as a stack-based code because stack-based codes are compact. This facilitates the development of different font engines on different platforms. However, this design makes it very difficult to reason about, or directly apply standard compiler optimizations to, TrueType bytecode. To solve this problem, we will next demonstrate the design of a new three-address code COI and describe the transformation process from TrueType to COI.

Figure 4.2: Function labels are implicit in TrueType

```
PUSH[ ]
2 1
FDEF[ ]
DUP[ ]
PUSH[ ]
1
ADD[ ]
RCVT[ ]
PUSH[ ]
3
CINDEX[ ]
DUP[ ]
SRP1[ ]
GC[0]
SUB[ ]
SWAP[ ]
RCVT[ ]
SWAP[ ]
SUB[ ]
SCFS[ ]
ENDF[ ]
FDEF[ ]
DUP[ ]
RCVT[ ]
RTG[ ]
ROUND[00]
WCVTP[ ]
ENDF[ ]
```

Figure 4.3: CALL instructions' destinations are implicit in TrueType bytecode

```
    IF[ ]
    PUSH[ ]
    16
    SCVTCI[ ]
    PUSH[ ]  /* 2 values pushed */
    22 0
    WS[ ]
    EIF[ ]
    CALL[ ]
    CALL[ ]
    CALL[ ]
```

Chapter 5

Design and Implementation

Figure 5.1 shows the steps we take for TrueType optimization and how all of our components fit together. First, we transform TrueType bytecode into COI, using results from the abstract interpreter. Second, we optimize the COI bytecode. Finally, (although this step is not implemented yet), we would transform COI back into TrueType.

5.1 Abstract Interpreter

5.1.1 Observations

Our design was influenced by two observations about TrueType bytecode, as mentioned in the previous chapter:

- Stack depth: due to the presence of loop instructions and implicitly-targetted function calls, calculating the current stack depth for every instruction is challenging.
- Data types: Although there are no explicit stack data types in TrueType syntax, TrueType instructions do use different types for each operand: integer, shortFrac, Fixed, FWord, uFWord, F2Ddot14 and longDateTime.

To analyze the behaviour of instructions in loops, we need the value of the loop state variable. Moreover, we need to accurately associate instructions with their operands, which are on the program stack. To gain all this information, we might consider implementing a TrueType interpreter. However, unlike a real font interpreter, at analysis time, we do not

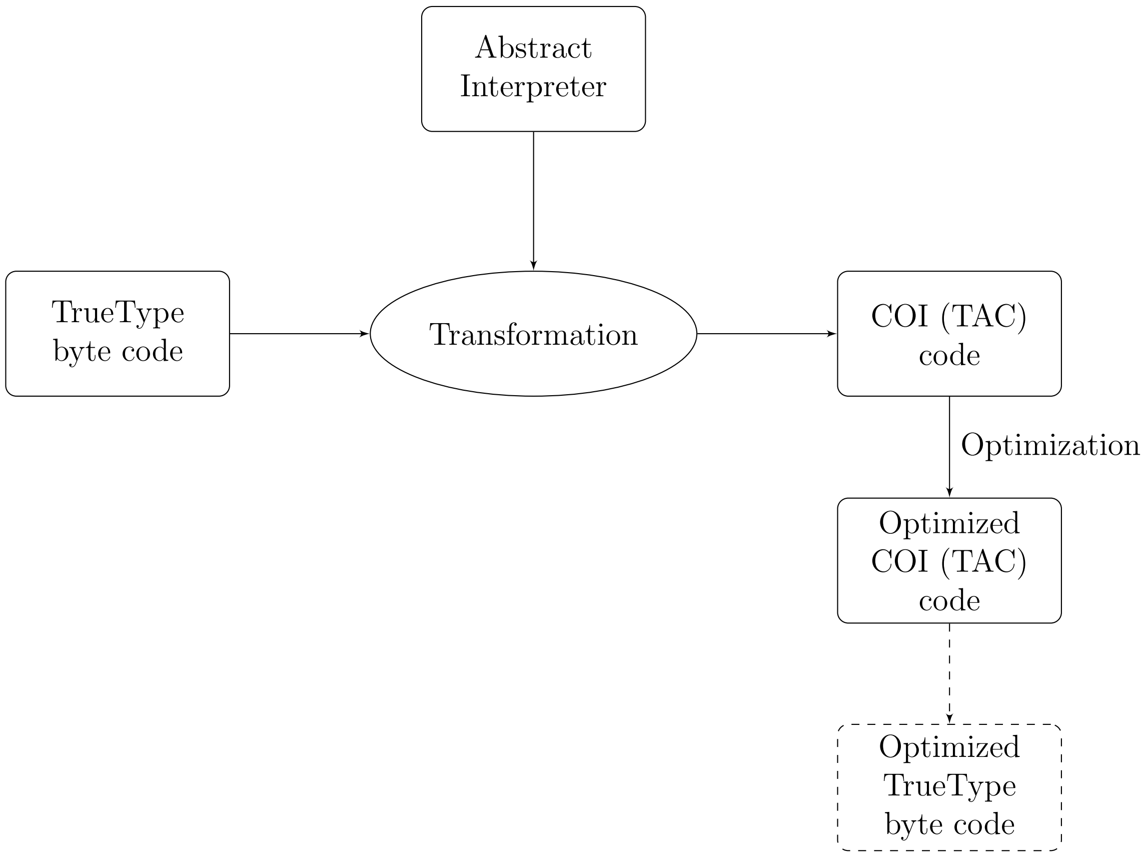

Figure 5.1: Relationships between COI components. Dashed box not yet implemented.

have run-time context, such as the target PPEM and the version number of the font scaler (engine). Moreover, we don't need all the data in program stack to be concrete—unlike a true font engine, we will not rasterize the font nor will we generate bitmap images for glyphs.

Futhermore, implementing a full TrueType interpreter requires a lot of work and it provides more information than we need for stack-based bytecode transformation. For example, we don't need the information about the zp0, zp1, zp2 variables in the graphics state, as they are only used in generating glyph images.

Our second observation provides guidance for differentiating unnecessary information from necessary information on the program stack. We only need to store and calculate concrete values for integer-typed data. We can use abstract values to represent other graphics-related data on the program stack.

We therefore use an abstract interpreter as a first stage in our analysis framework. Our interpreter introduces abstract values to represent data on the program stack.

For data which is either unknown (because it is program input) or associated with points and vectors, we create a set of abstract dataType classes, which we can put onto the stack in the place of normal values.

Consider, for instance, our interpretation of the **MPPEM** instruction. Since the PPEM is unknown, we push an object of the abstract data type class **PPEM_Y()** or **PPEM_X()**, representing the result, onto the program stack.

```
def exec_MPPEM(self):
        if self.graphics_state['pv'] == (0, 1):
            self.program_stack.append(dataType.PPEM_Y())
        else:
            self.program_stack.append(dataType.PPEM_X())
```

For concrete integer-typed values, we calculate a concrete result and push that result back onto the program stack.

If the operands of an expression contain both abstract values and concrete integer-typed values, we use an expression containing both the abstract value and the concrete value to represent the result, and push that result back onto the stack. Hence, data on our stack are expressions potentially containing both concrete values and abstract values.

In COI, all data on the stack get an associated variable name, or identifier. Then, in COI instructions, we use explicit variable identifiers instead of implicit stack indices to refer to this data. In this way, every operand of every COI instruction is explicit.

5.1.2 Type Inference

The TrueType manual provides type information for instructions, namely for their inputs (data pushed onto the stack) and outputs (data popped from the stack). Our abstract interpreter uses this information. In this section, we give an example and discuss how we infer type information for data on the stack.

Consider instruction **ROUND**. In the TrueType specification, the round algorithm is quite complex. However, the actual values are unnecessary for our purposes. In our **exec_ROUND** implementation, we simply use a F26Dot6 object to represent uninterpreted rounded values on the stack, as shown in Figure 5.2.

```
def exec_ROUND(self):
        self.program_stack_pop()
        self.program_stack.append(dataType.F26Dot6())
```

Figure 5.2: Implementation of ROUND in abstract interpreter

5.1.3 Accurate Stack Height

Even though there are abstract values on the stack, the stack height is always accurate. This is because the number of popped values and pushed values is specified for most instructions. The exception is loop-related instructions. Fortunately, in practice, we found that their effects on the stack height could also be accurately calculated by our abstract interpreter. Loops rely on the loop variable in graphics state, which we were able to always keep concrete.

5.1.4 Implementation

Our interpreter implementation contains two main steps. The first step is to tokenize plain strings in the TTX input into instructions and then construct a list of instruction instances. The second step is to structure the TrueType instruction list and build a successor list for every instruction.

For example, in Figure 5.3, the successors for the IF on line 3 are 1) the DUP on line 4 and 2) the POP on line 17. Similarly, the successors for the IF on line 10 are 1) the WCVTP on line 11 and 2) the POP on line 13.

ExecutionContext

We designed a class called **ExecutionContext** to represent the execution environment for every TrueType bytecode, including a program stack, a CVT table, a storage area, and a representation of the current graphics state.

We implemented a method exec_InstructionName for every TrueType instruction. This method modifies the current ExecutionContext with the effects of InstructionName.

Figure 5.3: TrueType bytecode code which includes IF-ELSE instructions

```
 MPPEM ( [ ] )
 LTEQ ( [ ] )
 IF ( [ ] )
     DUP ( [ ] )
     PUSH ( [ 3 ] )
     CINDEX ( [ ] )
     RCVT ( [ ] )
     ROUND ( [ 1 ] )
     GTEQ ( [ ] )
    IF ( [ ] )
        WCVTP ( [ ] )
    ELSE ( [ ] )
        POP ( [ ] )
        POP ( [ ] )
    EIF ( [ ] )
ELSE ( [ ] )
    POP ( [ ] )
    POP ( [ ] )
EIF ( [ ] )
```

Use stack to store back pointers

Since a program is a sequence of TrueType instructions, we process instructions one-by-one from the instruction stream. Because a normal interpreter only processes concrete values, it only executes one successor per instruction, depending on the value. This approach works because, in most cases, instructions have only one successor.

However, things get more complicated when we encounter function calls, since calls modify the flow of control. In addition, some instructions have more than one successor. As we may have abstract values on our stack, there may not be enough information for us to determine which successor should be executed. As a result, we sometimes need to traverse all possible branches.

We therefore implement a special method to simulate the program counter correctly.

Furthermore, when we are traversing nested if-else blocks, and execute a chain of function calls, we need to store more than one program counter. To track all program counter values, we use a stack to store back pointers, leveraging the stack last-in-first-out property. This enables us to correctly return to the last back pointer stored upon block end or return.

In the following, we will explain our treatment of function calls and if-else clauses.

- **Function Call** As discussed in Chapter 4, function labels are implicit in TrueType. We first preprocess the fpgm program and get a structured function table. Figure 5.4 shows an example of the TrueType bytecode after being preprocessed. The fpgm and endf instructions are completely removed and every function definition starts with its function label. In this way, function definitions become explicit, which also provides a concrete function table for implementing function call in later steps. Internally, the function table is a hash map structure that maps function labels to functions.

Figure 5.4: Example of Function Definitions After Preprocessing

```
Function #1
 DUP([])
 RCVT([])
 SWAP([])
 DUP([])
 PUSH([205])
 WCVTP([])
 SWAP([])
 DUP([])
 PUSH([346])
 LTEQ([])
 IF ([])
     SWAP([])
     DUP([])
     PUSH([141])
     WCVTP([])
     SWAP([])
 EIF ([])
 DUP([])
 PUSH([237])
 LTEQ([])
```

Consider a sequence such as `push[1] call[]`. When executing the `call`, we store the current program counter into **back_pointers**. Next, we look up the function labelled 1. Then, we move the program pointer to the beginning of the function. Once we finish executing the last instruction in function 1, we pop the program counter from **back_pointers**, returning the counter to the caller. This is analogous to the execution stack in usual programs.

- **If-Else Clause** Similarly to how we implement function calls, we also store the original program counter into **back_pointers**. Besides the program pointer itself, we also store the environment context associated with the program pointer. After we finish traversing one branch, we need to restore the program pointer, as well as the environment context, to traverse the other branch.

  After we traverse all branches, we then merge the branches' environment contexts. We have added assertions to our interpreter to enforce that the stack height of different branches must always be the same. This property is not required in the TrueType specification. However, we found that in our experiments, stack heights on different branches were indeed always the same.

  Although stack heights are the same, stack values may of course differ. In that case, we generate an OR expression to merge together results.

5.2 Translating TrueType to COI

Having preprocessed our bytecode and generated abstract values, the next step is to generate three-address code.

5.2.1 Properties of COI

Our three-address code, COI, has the following properties.

- COI is stack-less. Unlike TrueType bytecode, COI has completely removed the stack concept from the language definition.
- Every local variable is explicit and is initialized before use.
- The number of operands of every expression is no more than three. Instruction operands must be variables.

We next divide the TrueType instructions into different categories and discuss the translation from TrueType bytecode to equivalent COI instructions for every category.

5.2.2 Assignment Statements

We introduce the notion of variable in COI, which is absent from the TrueType bytecode, and introduce assignment statements to associate the data on the stack with variable names. In general, the main difficulties in analyzing TrueType bytecode are due to the fact that bytecode instructions affect the program stack, and thus have implicit uses of values on the stack. COI instructions can instead refer to data by variable identifier, eliminating implicit uses of values on the stack. This completely removes the complexities of calculating the actual stack depth from the bytecode analysis phase, making sophisticated analyses possible. After we have introduced assignment statements and variable names, our analyses can reason about data as identified by variable names, thus removing complications introduced by the stack. Listings 5.1 and 5.2 show before-and-after views of push instructions in TrueType and as converted to COI, respectively.

```
PUSH[ ]  /* 7 values pushed */
71 70 45 31 16 51 15 85 7
```

Listing 5.1: TrueType

```
prep0:=71
prep1:=70
prep2:=45
prep3:=31
prep4:=16
prep5:=51
prep6:=15
prep7:=85
prep8:=7
```

Listing 5.2: COI

5.2.3 Arithmetic and Logic Operations

We designed a group of arithmetic and logic operation statement in COI. It is straightforward to translate TrueType arithmetic and logic operations (all binary) into their COI equivalents. COI arithmetic and logic statements all have the same format **op3 = op1 op op2** where the right-hand side contains one operation and two operands and the left side has one variable. Table 5.1 shows the translation from TrueType arithmetic and logic operations to equivalent COI instructions.

TrueType Code	**COI Code**
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`ADD[]`	`v3 = v1 + v2`
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`SUB[]`	`v3 = v1 - v2`
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`DIV[]`	`v3 = v1 / v2`
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`MUL[]`	`v3 = v1 * v2`
`PUSH[]`	`v1 = 1`
`1 0`	`v2 = 0`
`AND[]`	`v3 = v1 AND v2`
`PUSH[]`	`v1 = 1`
`1 0`	`v2 = 0`
`OR[]`	`v3 = v1 OR v2`
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`EQ[]`	`v3 = v1 == v2`
`PUSH[]`	`v1 = 1`
`1 2`	`v2 = 2`
`NEQ[]`	`v3 = v1 != v2`

Table 5.1: COI Representations of TrueType Arithmetic and Logic Operations

5.2.4 IF/ELSE/EIF

To represent bytecode if-else clauses (which are, fortunately, structured), we designed an if-else clause in COI that explicitly represents conditions and includes pair braces. TrueType includes both if/endif and if/elseif/endif forms. We illustrate the conversion below.

- **TrueType Instruction: IF[]/EIF[]**

```
TrueType              COI

                      v1 = 1
 PUSH([1, 2])         v2 = 2
 RS([])               v3 = storage_area[v2]
 EQ([])               v4 = v3 == v1
 IF([])               if (v4) {
PUSH([2, 3])              v5 = 2
ADD[]                     v6 = 3
 EIF([])                  v7 = v5 + v6 }
PUSH[2,1]             v7 = 2
SUB[]                 v8 = 1
                      v9 = v8 - v7
```

- **TrueType Instruction: IF[]/ELSE[]/EIF[]**

```
TrueType              COI

                      v1 = 1
                      v2 = 2
PUSH([1, 2])          v3 = storage_area[v2]
 RS([])               v4 = v1==v3
 EQ([])               if (v4) {
 IF([])                   v8 = 2
    PUSH([2, 3]           v9 = 3
    ADD[]                 v10 = v8 + v9 }
 ELSE([])             else {
    PUSH([2, 1])          v5 = 2
    SUB[]                 v6 = 1
 EIF([])                  v7 = v5 - v6 }
 PUSH[3,1]            v10 = 3
 DIV[]                v11 = 1
                      v12 = v10/v11
```

5.2.5 Function call

The following is an example of a function call. Note that the target of the call is implicit in TrueType bytecode.

```
PUSH([83])
CALL[]
```

Assume that the function table contains a function labeled 83.

```
Function #83:
PUSH([18])
 SVTCA([0])
 MPPEM([])
 SVTCA([1])
 MPPEM([])
 EQ([])
 WS([])
```

The function call can then be translated into COI code:

```
prep19 = 83
CALL(prep19)
```

The function will be transformed into COI code:

```
BeginFunctionCALL:
fpgm0:=18
GS[freedom_vector]:=0
GS[projection_vector]:=0
fpgm1:=PPEM_Y
GS[freedom_vector]:=1
GS[projection_vector]:=1
fpgm2:=PPEM_X
fpgm3:=fpgm2 == fpgm1
storage_area[$fpgm0]=$fpgm3
EndFunctionCALL
```

5.2.6 Reads and writes from storage area and CVT table

These instructions can be straightforwardly translated into pseudo-assignment instructions.

- **RS[] Read Store**

  The RS[] can be translated into **v1 = storage_area[v2]**.

- **WS[] Write Store**

  The WS[] can be translated into **storage_area[v1] = v2**.

- **RCVT[] Read Control Value Table entry**

  The RCVT[] can be translated into **v1 = cvt_table[v2]**.

- **WCVTF[]/WCVTP[] Write Control Value Table in Funits/Pixel units**

  The WCVTF[] can be translated into **cvt_table[v1] = v2**.

5.2.7 Instructions that manage the stack

We use our abstract interpreter to calculate the current stack depth at each instruction in the TrueType program. Using that information, we can refer to the right variables in the corresponding COI instruction.

Some instructions simply modify the program stack's depth without introducing data or referring to any values from the stack. Examples include CLEAR[] and POP[]. We need not translate such instructions to COI; it suffices to manipulate the abstract stack depth in the interpreter.

- **CLEAR[] (CLEAR the stack)** This instruction clears the all data from the program stack, so that the program stack will be empty afterwards. We do not translate the CLEAR[] statement. We simply stop referring to the variables holding data that was cleared, and set the depth of the program variables' stack to 0 in the abstract interpreter.

- **DEPTH[] (DEPTH of the stack)** We get the current depth of the stack from the abstract interpreter, and assign that depth value to a variable. The DEPTH[] instruction thus gets translated into a variable assignment **v = SIZE**.

- **DUP[] (DUPlicate top stack element)** To translate the DUP statement, we create a new variable and assign to it the variable on top of the stack. For example:

TrueType	**COI**
`PUSH([2])`	`v1 = 2`
`DUP[]`	`v2 = v1`

- **POP[] (POP top stack element)**

  We do not translate the POP[] statement. We simply need to decrement the stack pointer in the interpreter state.

- **CINDEX[] (Copy INDEXed element)**

  As with DUP[], we also create a new variable and copy the indexed element in program stack. For example (the second indexed element in the stack is v1):

TrueType	**COI**
	`v1 = 3`
`PUSH([3,4,2])`	`v2 = 4`
`CINDEX[]`	`v3 = 2`
	`v4 = v1`

- **MINDEX[] (Move INDEXed element)**

  This instruction is similar to **CINDEX[]**.

TrueType	**COI**
	`v1 = 3`
`PUSH([3,4,2])`	`v2 = 4`
`MINDEX[]`	`v3 = 2`
	`v4 = v1`

- **ROLL[] (ROLL top 3 stack elements)**

  We use variable assignments to transform the ROLL[] instruction.

```
TrueType              COI

                      v1 = 3
                      v2 = 4
                      v3 = 2
PUSH([3,4,2])         $temp = v3
ROLL[]                v3 = v1
                      v1 = v2
                      v2 = $temp
```

- **SWAP[] (SWAP top two stack elements)**

```
TrueType              COI

                      v1 = 3
                      v2 = 4
PUSH([3,4])           $temp = v1
SWAP[]                v1 = v2
                      v2 = $temp
```

5.2.8 Graphics State Related Instructions

We convert graphics state instructions into pseudo-assignments, and include two examples.

- **SLOOP[] (Set LOOP variable)**

  SLOOP[] is translated into **GS[loop] = v1**.

- **SPVTCA[a] (Set Projection Vector To Coordinate Axis)**

  SPVTCA[a] is translated into **GS[projection_vector] = a**.

We hope that the loop variable remains concrete, since loop-related instructions use the loop variable to pop values from stack. For other graphics state, we do not need to keep values. We retain graphics state-related instructions in COI only to enable us to transform COI back to TrueType bytecode.

5.2.9 Other

Instructions in this category all map into (hard-coded COI) function calls with or without return values. This category include two categories of instructions:

1. Instructions that only pop data from stack and push nothing onto the stack. This kind of instruction only affects the height of the program stack, and typically has some other (drawing-related) side effect. We map instructions onto function calls without return values. For example, ALIGNPTS[] becomes ALIGNPTS(p1, p2).

2. Instructions that push one return result onto the stack. We map these instructions onto function calls with a return value. For example, we transform MAX[] to v3 = MAX(v1, v2), and MPPEM[] to v1 = MPPEM().

We transform each instruction concurrently with the abstract interpretation of the instruction. We push every new declared variable, along with its value, onto the variable stack. During instruction execution, we pop the corresponding variables from the stack.

5.3 Possible Optimizations

Transforming TrueType bytecode into COI allows for standard compiler optimizations. In this section, we list three example optimizations that could be applied to COI. Currently, we have implemented constant propagation for COI, and discuss results in chapter 6.

5.3.1 Constant propagation

Using constant propagation, we gained a complete set of function calls used in each *glyph program* and *control value program.*

5.3.2 Dead code elimination

After obtaining the set of functions used in glyph programs, we could perform dead code elimination the *font program* and remove the functions that are declared but never used.

5.3.3 Branch Elimination

If we could get all possible environmental input (for a fixed device), we could perform branch elimination and delete branches that will never be executed. Both device-specific and device-independent optimizations are possible. In the device-specific version, we would get a font file optimized for a certain device.

5.4 Experimental Methodology

We ensured the correctness of our implementation using several approaches.

- **Assertions in the Abstract Interpreter**

  Some invariants in TrueType bytecode can be translated into program assertions: for instance, keys for the CVT table and write storage indexed read are constant. Also, the key indexed value must have been previously written-to. At the end of execution, program stack height is always zero. There must never be a stack underflow.

  We added the above assertions into the implementation of the abstract interpreter, ensuring that these invariants always hold during program execution.

- **Create test cases for every instruction**

  We manually created a suite of test cases to test the result of transforming every TrueType instruction into COI. This helps to verify the correctness of our implementation of the transformation process.

- **Manual Inspection**

  For each instruction execution, our abstract interpreter's logging will show how the instruction affects the program environment. We manually read the logging for every instruction to ensure the correct behavior of our abstract interpreter.

- **Test Against TrueType Interpreter** We instrumented the mature TrueType interpreter, FreeType [2], to print out target function labels for call sites that it encounters in TrueType bytecode. We then compared our experimental results in chapter 6 against the results from FreeType. This method helps ensures the correctness of our experimental results as reported in Chapter 6.

Chapter 6

Experimental Results

We used our tool to analyze several popular, deployed, font files belonging to the Google Noto Fonts suite[1], thus demonstrating the effectiveness of our work. Our experimental subjects are the NotoKufiArabic, NotoSansBoldItalic, NotoSansBold, and NotoSansItalic fonts. The sizes for NotoKufiArabic, NotoSansBoldItalic, NotoSansBold, and NotoSansItalic are 1.2 MB, 7.6 MB, 7.5 MB, and 7.8 MB separately.

Our tool successfully transformed all the glyph programs, control value programs and font programs of these font files into COI. We also performed constant propagation on the generated COI to compute the set of functions which are ever called from control value programs and from every glyph program.

Table 6.1 summarizes the experimental results, presenting statistics about the call graph, particularly the number of reachable functions from the control value program and from the glyph programs. Table A.1 shows the actual function call sites in each glyph in the font NotoKufiArabic. Functions that are (transitively) called in the control value program can be reached throughout the font file (labelled global in the table), since the control value program is executed before every glyph in the font. Functions that are called from glyph programs can only be reached locally (labeled local in the table). We have cross-checked these remarkable results by instrumenting the FreeType interpreter to report counts of reachable functions.

As shown in Table 6.1, NotoKufiArabic defines outlines for 392 glyphs and includes 90 functions in its font program. However, the control value program only calls 4 functions. Also, only 2 functions are called from any glyph programs for NotoKufiArabic. Table 6.1

[1]https://www.google.com/get/noto/

Table 6.1: Counts of reachable functions in selected Google Noto Sans fonts. Under 15% of bytecode functions are reachable.

Font Name	# Glyphs	# Functions	# Functions reachable globally	# Functions reachable locally	Functions reachable in total
KufiArabic	392	91	4	2	6
SansBoldItalic	2101	91	5	5	10
SansBold	2330	91	5	5	10
SansItalic	2111	91	5	6	11

further shows that only a very small fraction (from 6/91 to 11/91) of the functions in the function table are actually used in the other Google Noto fonts.

To better understand the use of functions in font files, we also calculate the number of static callsites to each function in our four subject fonts. Table 6.2 shows another view of the reachable functions information from Table 6.1. Glyph programs in every font tend to call certain functions many times. In NotoKufiArabic, there are 258 call sites which invoke function #89. In NotoSansBoldItalic, the numbers of call sites of function #38, #72 and #73 are 598, 222, and 596.

We speculate that the large amounts of unused code might result from automatic hinting tools: producing a high-quality hint program requires both artistry and programming knowledge. Font designers appear to often use existing libraries of hint programs.

Appendix A presents our experimental results in more detail. It shows the exact function label invoked at each call site of the Control Value Program and in glyph programs. Given this information, font subsetting tools will be able to determine which functions are used and which functions are not. This information permits the removal of unused functions directly from the font files by existing font manipulation tools.

Our experimental results provide helpful information usable by font subsetting tools in their efforts to remove unused instructions. We have also found that implementing the constant propagation was aided by the use of COI. The three-address code property of COI enables optimization for TrueType bytecode.

Table 6.2: Counts of static callers to functions are concentrated for the KufiArabic, SansBoldItalic, SansBold, and SansItalic fonts.

function label	# callers KufiArabic	# callers SansBoldItalic	# callers SansBold	# callers SansItalic
14		1		
31 (global)	1	11	14	13
37			930	
38		595	224	593
62		89		94
70 (global)		9	3	2
72	12	222	202	159
73		596	59	250
83 (global)	1	1	1	1
84 (global)	1	1	1	1
85 (global)	2	35	31	35
89	258	1476	1432	1482
91		4		4

Chapter 7

Conclusions and Future Work

7.1 Discussion

Our work takes a significant step towards practical TrueType bytecode analysis and optimization. COI is a novel three-address code specifically for TrueType. Our implementation successfully translates Truetype bytecode into COI and makes the implict control flow graph explicit. Our experimental results show that our work can sucessfully detect unused code in real-life font files.

A notable feature of COI's design is the use of an abstract interpreter to associate the data in the implicit program stack with corresponding instructions. COI's three-address property enables optimizations on TrueType bytecode.

7.2 Limitations

Although there are no standard loop instructions in TrueType (some instructions repeat for a number of iterations as specified by SLOOP), TrueType does include some jump instructions. Our framework does not currently support the following instructions: JROF[] (Jump Relative On False), JROT[] (Jump Relative On True) and JMPR[] (JuMP Relative). We have never observed any uses of these instructions in font files in the wild, and no fonts in our experiments contain those instructions.

7.3 Future Work

The bulk of the future work consists of implementing a translation from COI to TrueType bytecode. Currently, our framework can only be used to optimize TrueType bytecode. One direction for the future work would be to support more formats that are similar to TrueType, such as OpenType and Postscript.

APPENDICES

Appendix A

Experiment Result in detail

The following table presents a mapping from glyph id to the call sites in that glyph program. Table A.1 and Table A.2 show function labels called in the control value program and glyph program in NotoKufiArabic and NotoSansBold. We didn't show the result of every glyph program due to the large amount of glyphs in the font files. We presented information for selected glyph programs with more than two call sites.

Table A.1: Functions Called from Selected Glyph Programs of NotoKufiArabic

glyph id	calls functions
prep (Control Value Program)	[83, 84, 85, 85, 31]
glyf.uni06AB.medi	[89, 89]
glyf.uni0637.medi	[89, 89]
glyf.uni066A	[89, 89]
glyf.uni066F	[89, 89, 89]
glyf.uni076A.init	[89, 89, 89]
glyf.uni0665	[89, 89]
glyf.uni0669	[89, 89]
glyf.feh_dotless.isol	[89, 89]
glyf.hah_alt.fina	[89, 89, 89]
glyf.uni062D.fina	[89, 89, 89, 89]
glyf.uni06A9.fina	[89, 89]
glyf.uni0645.init	[89, 89]
glyf.uni06C1.fina	[89, 89]

glyf.uni0633.fina	[89, 89]
glyf.uni06AB.fina	[89, 89]
glyf.uni06C5.fina	[89, 89, 89]
glyf.uni06C1.medi	[89, 89, 89]
glyf.uni0639	[89, 89]
glyf.uni0637	[89, 89]
glyf.uni0635	[89, 89, 89]
glyf.uni0633	[89, 89]
glyf.uni06A1.fina	[89, 89]
glyf.uni0663	[89, 89]
glyf.uni075B.fina	[89, 89, 89]
glyf.uni0635.init	[89, 89]
glyf.uni06AA.init	[89, 89]
glyf.uni06CD.fina	[89, 89, 89]
glyf.uni06A9	[89, 89]
glyf.allah	[89, 89]
glyf.uni066F.fina	[89, 89, 89, 89]
glyf.uni06AA	[89, 89]
glyf.uni06AB	[89, 89]
glyf.uni0639.init	[89, 89]
glyf.uni0647.init	[89, 89, 89]
glyf.uni062D.medi	[89, 89]
glyf.uni06AA.fina	[89, 89]
glyf.uni066F.medi	[89, 89]
glyf.uni0645.fina	[89, 89, 89]
glyf.uni0639.medi	[89, 89, 89]
glyf.uni06C1.init	[89, 89]
glyf.uni060F	[89, 89, 89]
glyf.uni060E	[89, 89]
glyf.uni06BE	[89, 89, 89]
glyf.uni06BE.fina	[89, 89, 89]
glyf.uni0602	[89, 89]
glyf.uni0603	[89, 89]
glyf.uni0601	[89, 89, 89, 89, 89]
glyf.uni06E9	[72, 72]
glyf.uni076A.fina	[89, 89, 89, 89]
glyf.uni0648.fina	[89, 89, 89, 89]
glyf.riyal	[89, 89, 89]

glyf.uni06AA.medi	[89, 89]
glyf.uni0635.medi	[89, 89]
glyf.uni0649.fina	[89, 89, 89]
glyf.uni076A	[89, 89, 89]
glyf.uni062D.init	[89, 89]
glyf.uni06F4	[89, 89]
glyf.uni06F5	[89, 89, 89]
glyf.uni0648	[89, 89, 89]
glyf.uni0649	[89, 89]
glyf.uni0647	[89, 89]
glyf.uni0645	[89, 89]
glyf.uni0643	[89, 89, 72, 89]
glyf.uni0647.fina	[89, 89]
glyf.uni0637.fina	[89, 89]
glyf.uni0631.fina	[89, 89]
glyf.uni0643.fina	[89, 89, 72, 89]
glyf.uni0639.fina	[89, 89, 89]
glyf.hah_alt.isol	[89, 89]
glyf.uni0647.medi	[89, 89, 89]
glyf.uni0645.medi	[89, 89, 89]
glyf.uni0635.fina	[89, 89, 89]
glyf.uni06D2.fina	[89, 89]
glyf.uni06C5	[89, 89]
glyf.uni06C4	[89, 89, 72, 89]
glyf.uni06C4.fina	[89, 89, 72, 89, 89]
glyf.uni075B	[89, 89]
glyf.uni066F.init	[89, 89]
glyf.uni06CD	[89, 89]
glyf.uni0644.fina	[89, 89]
glyf.uni062D	[89, 89, 89]
glyf.uni062F	[89, 89]
glyf.uni076A0627.isol	[89, 89, 89]
glyf.uni076A0627.fina	[89, 89, 89]
glyf.uni06BA.fina	[89, 89]
glyf.uni0621	[89, 89, 89]
glyf.uni0643.medi	[89, 89]
glyf.uni0643.init	[89, 89]
glyf.uni076A.medi	[89, 89, 89]

glyf.uni0637.init	[89, 89]
glyf.uni06AB.init	[89, 89]

Table A.2: Functions Called from Selected Glyph Programs of NotoSansBold

glyph id	calls functions
prep	[83, 84, 85, 85 (x5), 31 (x5), 70, 70, 85, 85, 85, 31 (x4), 85, 85, 31 (x5), 85, 85, 85, 70, 85 (x17)]
glyf.uni1D6D	[89, 89, 72]
glyf.uni1D6C	[89, 89, 72]
glyf.uni0409	[89, 89, 89, 89]
glyf.uni0402	[89, 89, 89]
glyf.uni0404	[89, 89, 73, 72, 73, 89]
glyf.six	[89, 89, 89]
glyf.uni0276	[89, 89, 89, 89, 89]
glyf.uni040A	[89, 89, 89]
glyf.uni1D1F	[89, 89, 89]
glyf.uni1D14	[89, 89, 89, 89, 72, 72, 89]
glyf.uni20A8	[89, 89, 89, 89]
glyf.lira	[89, 89, 89, 89]
glyf.uni1D6B	[89, 89, 89, 89]
glyf.uni1E2D	[37, 72, 72]
glyf.uni1E2C	[37, 72, 72]
glyf.Xi	[89, 89, 73, 89]
glyf.uni0411	[89, 89, 89]
glyf.uni0417	[89, 89, 73, 89]
glyf.uni04FB	[89, 89, 89, 89]
glyf.uni023B	[89, 89, 72]
glyf.uni023F	[89, 89, 89]
glyf.uni023D	[89, 73, 89]
glyf.uni042D	[89, 89, 73, 72, 73, 89]
glyf.uni0236	[89, 89, 89]
glyf.uni0235	[89, 89, 89]
glyf.uni20B9	[89, 89, 89]

glyf.uni20B4	[89, 89, 89, 89]
glyf.uni20B5	[89, 89, 72]
glyf.uni20B1	[89, 89, 89, 89]
glyf.uni20B2	[89, 89, 72, 72, 89]
glyf.five	[89, 89, 89]
glyf.peseta	[89, 89, 89, 89]
glyf.uni1D03	[89, 89, 72, 89]
glyf.uni1D02	[89, 89, 89, 89, 72, 72, 89, 72, 89]
glyf.uni1D01	[89, 89, 89, 72, 89]
glyf.uni1D07	[89, 89, 72, 89]
glyf.uni1D06	[89, 89, 72, 89]
glyf.uni1D08	[89, 89, 89]
glyf.uni01C4	[37, 37, 38]
glyf.uni1E39	[37, 37, 38]
glyf.uni1E38	[37, 37, 38]
glyf.uni1ED8	[37, 37, 38]
glyf.uni1ED6	[89, 89, 72, 72]
glyf.uni1E68	[37, 37, 38]
glyf.uni0335	[72, 72, 72]
glyf.oe	[89, 89, 89, 89, 89]
glyf.uni042E	[89, 89, 89]
glyf.uni042A	[89, 89, 89]
glyf.florin	[89, 89, 89]
glyf.uni01BD	[89, 89, 89]
glyf.uni01BC	[89, 89, 89]
glyf.uni01BB	[89, 89, 89]
glyf.uni01BA	[89, 89, 89, 89]
glyf.uni01B8	[89, 89, 89]
glyf.uni01B6	[89, 89, 89]
glyf.uni01B5	[89, 89, 73, 89]
glyf.uni1E08	[37, 37, 38]
glyf.uni04A9	[89, 89, 89, 72, 89]
glyf.uni04A8	[89, 89, 89, 89, 72, 89]
glyf.uni04A3	[89, 73, 72, 89]
glyf.uni04A7	[89, 89, 89]
glyf.uni04A6	[89, 89, 89]
glyf.uni04A5	[89, 73, 72, 89]
glyf.uni043D	[73, 72, 89]

glyf.uni0431	[89, 89, 89]
glyf.uni0432	[89, 89, 89]
glyf.uni0437	[89, 89, 89]
glyf.uni205E	[89, 89, 89, 89]
glyf.uni021D	[89, 89, 89]
glyf.uni1E1C	[37, 37, 38]
glyf.uni1E1B	[37, 72, 72]
glyf.uni01AB	[89, 89, 89]
glyf.uni01AA	[89, 89, 89]
glyf.tbar	[89, 89, 89]
glyf.uni1D23	[89, 89, 89]
glyf.uni04BF	[89, 89, 89]
glyf.uni04BD	[89, 89, 89]
glyf.uni04BE	[89, 89, 89]
glyf.uni04BC	[89, 89, 89]
glyf.uni01B9	[89, 89, 89]
glyf.uni20A0	[89, 89, 89, 89]
glyf.uni20A2	[89, 89, 89]
glyf.uni20AE	[89, 72, 72, 72]
glyf.uni20AF	[89, 89, 89, 89]
glyf.partialdiff	[89, 89, 89]
glyf.uni04C5	[89, 89, 89]
glyf.uni04C6	[89, 89, 89]
glyf.uni04C8	[89, 73, 72, 89]
glyf.glyph02370	[72, 72, 72]
glyf.uni04CA	[89, 73, 72, 89]
glyf.uni02A9	[89, 89, 89, 89]
glyf.uni02A8	[89, 89, 89, 89, 89]
glyf.uni02A0	[89, 89, 89]
glyf.uni02A3	[89, 89, 89, 89]
glyf.uni02A5	[89, 89, 89, 89, 89]
glyf.uni02A4	[89, 89, 89, 89, 89]
glyf.uni02A7	[89, 89, 89, 89]
glyf.uni02A6	[89, 89, 89]
glyf.e	[89, 89, 89]
glyf.uni02AF	[89, 89, 89]
glyf.E	[89, 89, 73, 89]
glyf.F	[89, 73, 89]

glyf.G	[89, 89, 89]
glyf.B	[89, 89, 73, 73, 89]
glyf.g	[89, 89, 89]
glyf.a	[89, 89, 89]
glyf.uni1D1E	[89, 89, 72, 72]
glyf.uni03EC	[89, 89, 89]
glyf.theta	[89, 89, 73, 89]
glyf.uni1E75	[37, 72, 72]
glyf.uni1E74	[37, 72, 72]
glyf.nine	[89, 89, 89]
glyf.OE	[89, 89, 89, 89, 73, 89]
glyf.glyph02366	[89, 89, 89]
glyf.glyph02367	[72, 72, 72]
glyf.glyph02368	[72, 72, 72]
glyf.glyph02369	[72, 72, 72]
glyf.Euro	[89, 89, 89, 89]
glyf.alpha	[89, 89, 89]
glyf.uni03DC	[89, 73, 89]
glyf.uni03DE	[89, 89, 73, 89]
glyf.uni048C	[89, 73, 89, 89]
glyf.uni048D	[89, 73, 89, 89]
glyf.uni03D0	[89, 89, 89, 72, 89]
glyf.uni03D7	[89, 89, 89]
glyf.uni03ED	[89, 89, 89]
glyf.eth	[89, 89, 89, 89]
glyf.sterling	[89, 89, 89]
glyf.uni0495	[89, 89, 89]
glyf.uni0494	[89, 89, 89]
glyf.theta1	[89, 89, 89, 89]
glyf.uni01E4	[89, 89, 72, 89, 89]
glyf.uni01E5	[89, 89, 72, 89, 89]
glyf.uni01EC	[37, 37, 38]
glyf.uni1E5C	[37, 37, 38]
glyf.Eth	[89, 89, 89]
glyf.ae	[89, 89, 89, 89, 89, 89]
glyf.dcroat	[89, 89, 89]
glyf.uni03FC	[89, 89, 89]
glyf.uni01DD	[89, 89, 89]

glyf.uni1D99	[89, 89, 89]
glyf.uni1D90	[89, 89, 89, 89]
glyf.uni1D91	[89, 89, 89, 89]
glyf.uni1D92	[89, 89, 89, 89]
glyf.uni1D93	[89, 89, 89, 89]
glyf.uni1D94	[89, 89, 89, 89]
glyf.uni1D95	[89, 89, 89, 89]
glyf.uni1D96	[89, 89, 89]
glyf.uni1D97	[89, 89, 89]
glyf.uni1D9A	[89, 89, 89, 89]
glyf.Aringacute	[89, 72, 72]
glyf.epsilon	[89, 89, 89]
glyf.franc	[89, 89, 89]
glyf.uni1D8B	[89, 89, 89, 89]
glyf.uni1D8A	[89, 89, 89]
glyf.uni1D8F	[89, 89, 89, 89, 89]
glyf.uni1D8E	[89, 89, 89]
glyf.uni1D89	[89, 89, 89]
glyf.uni1D88	[89, 89, 89]
glyf.uni1D83	[89, 89, 89, 89, 89]
glyf.uni1D82	[89, 89, 89, 89]
glyf.uni1D81	[89, 89, 89, 89]
glyf.uni1D80	[89, 89, 89]
glyf.uni1D87	[89, 89, 89]
glyf.uni1D86	[89, 89, 89]
glyf.uni1E1A	[37, 72, 72]
glyf.uni01A5	[89, 89, 89]
glyf.uni2116	[72, 89, 89]
glyf.uni1EB6	[37, 37, 38]
glyf.Theta	[89, 89, 73, 89]
glyf.uni01AD	[89, 89, 89]
glyf.uni0286	[89, 89, 89]
glyf.uni0284	[89, 89, 89]
glyf.uni0282	[89, 89, 89]
glyf.uni04E8	[89, 89, 73, 72, 89]
glyf.uni0193	[89, 89, 89, 89]
glyf.uni0190	[89, 89, 73, 89]
glyf.uni0191	[89, 89, 73, 89]

glyf.uni0197	[89, 89, 73, 89]
glyf.uni044D	[89, 89, 72, 72, 89]
glyf.uni044E	[89, 89, 73, 72, 89]
glyf.uni044A	[89, 89, 73, 72, 89]
glyf.uni044B	[89, 73, 72, 89]
glyf.uni044C	[89, 73, 72, 89]
glyf.beta	[89, 89, 89]
glyf.uni0267	[89, 89, 89]
glyf.uni0260	[89, 89, 89, 89]
glyf.uni0261	[89, 89, 89]
glyf.uni0262	[89, 89, 89]
glyf.uni026E	[89, 89, 89]
glyf.aringacute	[89, 89, 89, 72]
glyf.uni1EC6	[37, 37, 38]
glyf.uni1EC4	[89, 89, 73, 89, 72, 72]
glyf.uni0523	[89, 89, 73, 72, 89]
glyf.uni0522	[89, 89, 89]
glyf.uni0521	[89, 89, 89, 89]
glyf.uni0520	[89, 89, 89, 89]
glyf.uni045A	[89, 73, 72, 89, 89]
glyf.uni029A	[89, 89, 89]
glyf.uni029B	[89, 89, 89, 89]
glyf.uni029D	[89, 89, 89]
glyf.uni0291	[89, 89, 89]
glyf.uni0290	[89, 89, 89]
glyf.uni0293	[89, 89, 89, 89]
glyf.uni0454	[89, 89, 72, 72, 89]
glyf.uni0452	[89, 89, 89]
glyf.uni0459	[89, 89, 89, 73, 72, 89]
glyf.uni0188	[89, 89, 89]
glyf.uni0187	[89, 89, 89]
glyf.uni0181	[89, 89, 73, 73, 89]
glyf.uni0180	[89, 89, 89]
glyf.uni0183	[89, 89, 89]
glyf.uni2C76	[73, 72, 89]
glyf.uni018E	[89, 89, 73, 89]
glyf.uni018C	[89, 89, 89]
glyf.uni018B	[89, 89, 89]

glyf.uni1D98	[89, 89, 89]
glyf.uni1E9E	[89, 89, 89]
glyf.AE	[89, 89, 89, 73, 89]
glyf.uni0512	[89, 89, 89, 89]
glyf.uni0513	[89, 89, 89, 89]
glyf.uni0510	[89, 89, 73, 89]
glyf.uni0518	[89, 89, 89, 73, 89]
glyf.uni0519	[89, 89, 89, 89, 89]
glyf.uni04D8	[89, 89, 89]
glyf.uni04D9	[89, 89, 89]
glyf.uni0462	[89, 89, 89]
glyf.uni0463	[89, 89, 73, 72, 89]
glyf.uni0466	[73, 73, 89]
glyf.uni0464	[89, 89, 89]
glyf.uni0465	[89, 89, 73, 72, 89]
glyf.uni0469	[73, 72, 89, 89]
glyf.uni046B	[89, 72, 89]
glyf.uni046C	[89, 89, 89]
glyf.uni046F	[89, 89, 89, 89]
glyf.uni046D	[89, 73, 72, 89, 89, 72]
glyf.uni046E	[89, 89, 73, 89, 89]
glyf.uni2C65	[89, 89, 89]
glyf.uni2C64	[89, 89, 89]
glyf.uni2C63	[89, 89, 72, 89]
glyf.uni2C60	[89, 89, 89]
glyf.uni0248	[89, 73, 89]
glyf.uni0249	[89, 89, 89]
glyf.uni0243	[89, 89, 73, 89]
glyf.uni0240	[89, 89, 89]
glyf.uni0246	[89, 89, 73, 89]
glyf.uni0247	[89, 89, 89]
glyf.uni024B	[89, 89, 89]
glyf.uni024A	[89, 89, 89]
glyf.uni1D7C	[89, 72, 89]
glyf.uni1D7A	[89, 89, 89]
glyf.uni1D7F	[89, 89, 89]
glyf.uni1D7D	[89, 89, 72, 89]
glyf.uni1D79	[89, 89, 89]

glyf.uni1D77	[89, 89, 89]
glyf.uni1EAA	[89, 72, 72]
glyf.uni1EAC	[37, 37, 38]
glyf.uni0503	[89, 89, 89]
glyf.uni0505	[89, 89, 89]
glyf.uni0504	[89, 89, 89]
glyf.uni0507	[89, 89, 89]
glyf.uni0506	[89, 89, 89]
glyf.uni0509	[89, 89, 89]
glyf.uni04E1	[89, 89, 89]
glyf.uni04E0	[89, 89, 89]
glyf.uni04E9	[89, 89, 72, 72, 89]
glyf.uni0478	[89, 89, 89]
glyf.uni0473	[89, 89, 72, 72, 89]
glyf.uni0472	[89, 89, 73, 72, 89]
glyf.uni050C	[89, 89, 89]
glyf.uni050B	[89, 73, 72, 89]
glyf.uni050D	[89, 89, 89]
glyf.uni025E	[89, 89, 89]
glyf.uni025D	[89, 89, 89, 89]
glyf.uni025A	[89, 89, 89]
glyf.uni0255	[89, 89, 89]
glyf.uni0257	[89, 89, 89]
glyf.uni0256	[89, 89, 89]
glyf.uni0250	[89, 89, 89]
glyf.uni0253	[89, 89, 89]
glyf.uni0258	[89, 89, 89]
glyf.three	[89, 89, 73, 89]

References

[1] The fonttools project. https://github.com/behdad/fonttools/.

[2] The freetype project. http://www.freetype.org//.

[3] The itype project. http://www.monotype.com/services/screen-imaging-solutions/itype.

[4] Wolfram Amme, Thomas S. Heinze, and Jeffery Von Ronne. Intermediate representations of mobile code, 2007.

[5] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[6] Microsoft Corporation. Microsoft typography. https://www.microsoft.com/typography/default.mspx/. Accessed: 2014-12-5.

[7] Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. *SIGPLAN Not.*, 49(10):1–16, October 2014.

[8] Laurie J Hendren, Bhama Sridharan, V Sreedhar, and Y Wong. The simple ast-mccat compiler. *Design Notes*, 36, 1992.

[9] Roger D. Hersch. Character generation under grid constraints. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 243–252, New York, NY, USA, 1987. ACM.

[10] Roger D. Hersch and Claude Betrisey. Model-based matching and hinting of fonts. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 71–80, New York, NY, USA, 1991. ACM.

[11] Apple Inc. Truetype reference manual. https://developer.apple.com/fonts/TrueType-Reference-Manual/. Accessed: 2014-12-5.

[12] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[13] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software: Practice and Experience*, 20(12):1241–1258, 1990.

[14] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–179, 2003.

[15] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 41–52, New York, NY, USA, 2014. ACM.

[16] Ariel Shamir. Constraint-based approach for automatic hinting of digital typefaces. *ACM Trans. Graph.*, 22(2):131–151, April 2003.

[17] Beat Stamm. Visual true type: A graphical method for authoring font intelligence. In *Proceedings of the 7th International Conference on Electronic Publishing, Held Jointly with the 4th International Conference on Raster Imaging and Digital Typography: Electronic Publishing, Artistic Imaging, and Digital Typography*, EP '98/RIDT '98, pages 77–92, London, UK, UK, 1998. Springer-Verlag.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[19] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[20] Douglas E. Zongker, Geraldine Wade, and David H. Salesin. Example-based hinting of true type fonts. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 411–416, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.