Draft of April 17, 2014, at 2:04 am

Pomona College
Department of Computer Science

StackExplorer: Visualizing the Call Stack for Novice Programmers

Kimberly Merrill

April 17, 2014

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Rett Bull

Draft of April 17, 2014, at 2:04 am

Abstract

Novice programmers first encounter machine-level constructs in introductory computer architecture courses. Translating their knowledge of high-level concepts to the operations performed by the machine is a cognitively heavy conceptual jump as students become accountable for their program's data on the stack and explicit CPU state. The field of program visualization has proposed many models for visualizing challenging program execution concepts but has most extensively focused on high-level introductory programming concepts and low-level full-machine simulators.

StackExplorer was designed as a dynamic program visualization tool to reinforce the stack discipline as taught in existing introductory computer architecture courses. It represents the execution model as a series of stack frames and the effects the source code has on the contents of each frame, allowing users to step through C programs and examine the corresponding assembly instructions, call stack, and stack frames.

Acquiring the data needed for visualization from the host machine in step with the user's actions, StackExplorer's stack frame structure depends on the calling conventions and optimizations of the host architecture, prone to presenting non-idealized stack frames when run on programs with certain function patterns. Continued development of the tool should focus on identifying applications in which the architecture constraint does not conflict with learning goals and on evaluating the tool's pedagogical value through user testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To the novice programmer, computer programs operate as static entities. As the student develops fluency with a high-level programming language, she concurrently develops a mental model of program execution, often in which a program is source code and the all-encompassing computer is a magic box that takes in text and spits out results. This mental model defines how students approach programming.

> *Overall, what happens in programming is that the programmer constructs a mental model, an internal representation of their understanding of a programs intent, its data and execution, and applies this model to the code to make predictions and formulate changes.*
>
> Juha Helminen & Lauri Malmi [HM10]

When these mental models are static or otherwise flawed, a student artificially limits her capacity to effectively write, trace, and iterate on code.

Mapping the static text of a program to the dynamic process of its runtime execution is one of the core conceptual hurdles in introductory computer science pedagogies [Guo13]. First introductions to computer architecture seek to intervene in the formation of these fundamental misconceptions, stripping away layers of abstraction and making students accountable for their program data, their location on the stack, and their interpretation. In efforts to upset the complacency students develop while working with high level programming languages, introductory computer architecture lessons work to present a new notional machine that emphasizes the stack discipline and how real machines operate at a low level. This conceptual leap from a static notional machine to a dynamic notional machine, however, proves challenging for novice programmers.

According to Sorva, a notional machine is a characterization of software and hardware capabilities that "are abstract but sufficiently detailed, for a certain context, to explain how programs get executed and what the relationship of programming language commands is to such executions." As a model of how a computer works, notional machines are often implicit in computer science curriculums, allowing room for students to, unsupervised, establish contradictory or inadequate mental models that lead to later challenges in developing and debugging their programs [Sor13]. Many of these challenges arise from misconceptions in relation to "the hidden runtime world of the notional machine," or concepts that are not clearly visible in program code, because students are unable to associate the machine operations with the text on their screen [SKM13].

[Sor13] suggests pressing students to confront and consider how they reason about program execution in order to combat students' reliance on inadequate mental models. To concretize the desired notional machine, educators often propose the use of visualization tools, which provide animated views of program execution, to encourage students to think about programs dynamically and to provide an explicit representation of the computers operation. Novice programmers grapple with maintaining complete status representations of the machine's state - variables, references, function activations, and so forth - while tracing through programs because they are inexperienced at “selecting the right moving parts” [Sor13]. Thus the support of visualization tools allows students to validate their mental status representations and identify the causes of their misreading.

The field of educational program visualization focuses on high-level visual metaphors for fundamental programming constructs like objects and function calls and on heavy low-level constructs like the memory bus or hardware registers. Few existing program visualization tools straddle the divide between the high- and low- level and work to link static source code to basic machine-level execution.

Research into common programming misconceptions has shown that novice programmers consistently struggle with this middle ground, particularly with respect to the notion of stack frames, their purpose, and how they relate to the variables that are in scope within a function call [SS12]. The ACM Computer Science Curriculum emphasizes the importance of stack frame and subroutine call comprehension at both the high-level - for example, its relation to recursion - and the low-level - for example, the locations of variables on the stack and the movement of values into and out of registers - suggesting its centrality in the growth of students as computer scientists [SB10].

The purpose of this research is to construct a dynamic program visualization tool that facilitates the conceptual leap from high-level programming languages to machine-level program execution by visualizing the call stack at both the high- and low- level. By providing an interface for novice programmers to explore the direct effects their C programs have on the state of the machine, StackExplorer aids students in drawing connections between the code they write and the operations the machine performs, hybridized in its presentation of both low-level information such as memory addresses and high-level information such as variable assignment.

The primary focus will be presenting a tool for real-time step-by-step examination of program execution, thus eliminating the need for preliminary user input and allowing for flexibility of use so that students can visualize how an arbitrary C program operates on a given architecture. Through interactive execution snapshots, StackExplorer seeks to concretize a wholly dynamic notional machine consistent with the models taught in existing curriculums such that its visualizations correct inadequate mental models of program execution and reinforce the representations presented in introductory computer architecture courses.

Chapter 2

Background

2.1 Preface

Educators have adopted pedagogical applications of program visualization in computer science curriculums for topics ranging from data structures to compiler design. Although program visualization is also widely used for debugging and performance analysis applications, we consider only systems designed for educational purposes. These visualization tools endeavor to help students build more complete mental models of program behavior by displaying visual representations of data, relationships in the code, and corresponding actions taken by the computer.

Meta-analyses of program visualization in the classroom have shown mediocre results, demonstrating that student interaction with the visualization is the key predictor of learning outcomes rather than the tool's specific design considerations, with the educational value of entirely static visualizations or passive animations proven statistically insignificant [BS10]. Other studies have suggested that the benefits of visualization may be largely due to increased time spent on the task rather than the visualization itself, proposing that visualization tools can improve student engagement and encourage deeper exploration of programming concepts.

However, some studies have found that, when given visualization tools, novices rely on them, interacting heavily with the GUI but disregarding the code and choosing not to read the source before animating it [SKM13]. In their use of program visualization, instructors have reported seeing a "middle effect" wherein the weakest and strongest students considered visualization tools distracting, thus positively impacting the learning outcomes of only the students who fall in the in-between category [SKM13].

Here we consider four pedagogical tools and their contributions to the space of program visualization in regard to five main dimensions: content, flexibility, interactivity, accessibility, integration, and where data are available, effectiveness. Each tool's design implies a distinct notional machine for understanding program execution, abstracting away different details of the machine and emphasizing a specific set of relationships in its visualization content.

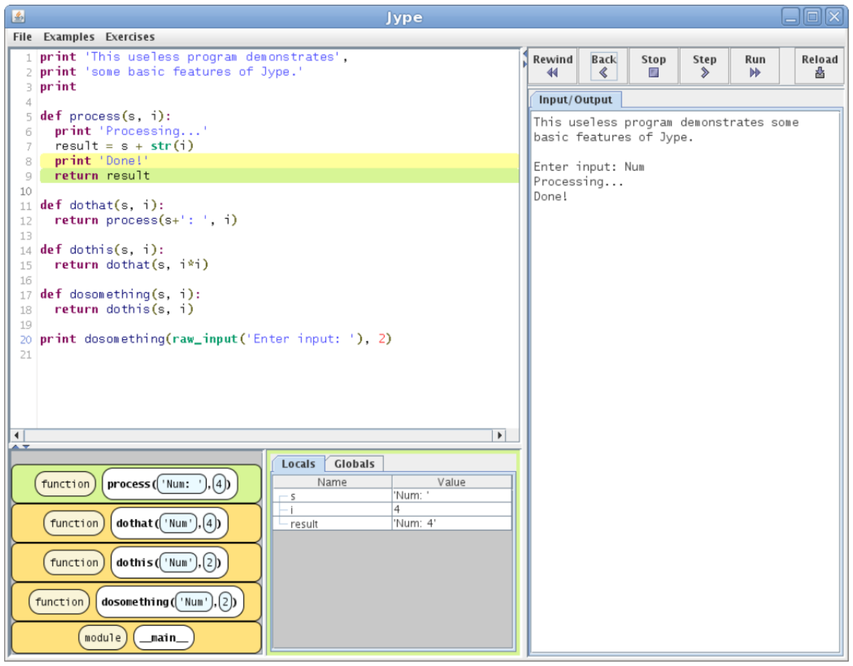
Within the five dimensions of evaluation, content refers to the construct or constructs being visualized and the form in which they are displayed to the user. Flexibility measures the level of generalizability in terms of input programs and the level of user control in what is displayed and what is hidden. Interactivity indicates the degree to which the user can explore and examine the visualization and accessibility is the cognitive load placed on the user in terms of the added learning curve in becoming comfortable with the tool. Integration refers to the consistency with existing notional machines in relation to how closely the visualization corresponds with and translates to in-use pedagogical models of the content. Where documented user testing exists, examination of the effectiveness on learning outcomes is also a useful descriptor of the tools design.

2.2 Jype

Researchers at the Aalto University School of Science and Technology developed Jype, an interactive web-based program visualization tool, to reinforce learning of basic code writing and program tracing at the CS1 level [HM10]. Jype provides an environment for unit-level Python programming, concentrating on higher-level concepts like variable assignment and data structures, and allows for controlled snapshot viewing wherein the user can run, step, and rewind through lines in the program. Its core notional machine interprets program execution as a series of function call containers with name-value variables stored within them.

Jype does not propose any abstract visual metaphors for programming constructs - for example, variables are a name and a value stored straightforwardly in a table - making the visualization's interpretation and use easily accessible to a novice programmer. It visualizes the call stack at a high level, with each function call separated out into a bordered box with the formal parameters substituted out for the actual parameters in the image, and allows the user to step back to the caller's scope by selecting a previous function call [HM10].

Extensions also allow for lists, trees, and other basic data structures to be visualized in their familiar forms, thus maintaining a high level of integration with existing conceptualizations of data structures and minimizing the amount of cognitive load on the user in following new representations of the same constructs. Details about these more complex structures are surfaced as tooltips so that the initial amount of displayed information is not overwhelming, giving users the option to zoom in on symbols when and where desired.



```
print 'This useless program demonstrates',
print 'some basic features of Jype.'
print

def process(s, i):
  print 'Processing...'
  result = s + str(i)
  print 'Done!'
  return result

def dothat(s, i):
  return process(s+': ', i)

def dothis(s, i):
  return dothat(s, i*i)

def dosomething(s, i):
  return dothis(s, i)

print dosomething(raw_input('Enter input: '), 2)
```

```
This useless program demonstrates some
basic features of Jype.

Enter input: Num
Processing...
Done!
```

Locals | Globals

Name	Value
s	'Num: '
i	4
result	'Num: 4'

Figure 2.1: Jype highlights the last executed source line in yellow to help users trace the control flow. [HM10]

Most notably, Jype distinguishes itself in its mode of user interaction, accepting content in the form of examples or exercises. With examples, instructors pre-create programs and explanatory notes for students to walk through, and with exercises, instructors provide a target code snippet or problem to solve and submit feedback to be displayed to the user as she works in the built-in text editor to code a correct solution [HM10]. Students can write and edit code inside the tool, making it highly flexible, but they must follow a structure pre-defined by the instructor, giving no flexibility in complete content ownership.

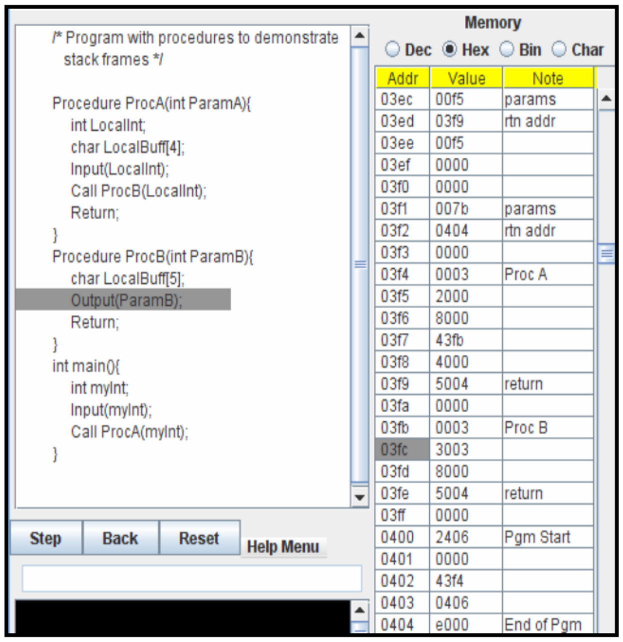
In qualitative studies, instructors found that students used Jype largely as a visual debugger while working through the in-environment exercises, using the automatic feedback written by the instructor - and surfaced in the tool - to adapt their code [HM10].

Jype integrates several strategies for promoting ease of use and high student engagement, representing concepts as consistent, textbook-style images and incorporating set lessons into the tool. Its implied notional machine is strictly limited in scope, however, abstracting away all messy middle- to low- level concepts and most of the call stack to better cater to its audience of first-time programmers. It is powerful in its presentation of very fundamental programming concepts, but does not draw the connection to the machine-level operations in its execution model, and with its preferred usage, can be characterized primarily as a novice-tailored visual debugger for instructor-designed mini programming assignments.

2.3 The Simple Machine Simulator

The notional machine proposed by Simple Machine Simulator (SMS) portrays program execution as a mapping from high-level instructions to memory. SMS was developed for use in a computer security course open to non-computer science majors with the goal, as the creators describe, of implementing the "minimum amount of detail required to teach the specific concepts of stack frames, function calls, and buffer overflows" [SB10]. As such, the guiding design principles focus on simplicity, demonstrating the stepwise execution of familiar C-style programs in terms of its effects on memory and abstracting away the lower level hardware and assembly constructs present in competing simulators.

To achieve simplicity and guarantee accessibility to students who have no experience with computer architecture, SMS comprises its own limited, fixed instruction set programming language that mimics basic C syntax. Its language supports only two basic data types - integers and strings, up to one parameter per subroutine, and seven basic high-level



```
/* Program with procedures to demonstrate
  stack frames */

Procedure ProcA(int ParamA){
    int LocalInt;
    char LocalBuff[4];
    Input(LocalInt);
    Call ProcB(LocalInt);
    Return;
}
Procedure ProcB(int ParamB){
    char LocalBuff[5];
    Output(ParamB);
    Return;
}
int main(){
    int myInt;
    Input(myInt);
    Call ProcA(myInt);
}
```

Addr	Value	Note
03ec	00f5	params
03ed	03f9	rtn addr
03ee	00f5	
03ef	0000	
03f0	0000	
03f1	007b	params
03f2	0404	rtn addr
03f3	0000	
03f4	0003	Proc A
03f5	2000	
03f6	8000	
03f7	43fb	
03f8	4000	
03f9	5004	return
03fa	0000	
03fb	0003	Proc B
03fc	3003	
03fd	8000	
03fe	5004	return
03ff	0000	
0400	2406	Pgm Start
0401	0000	
0402	43f4	
0403	0406	
0404	e000	End of Pgm

Figure 2.2: Proc A and Proc B indicate separate stack frames in SMS' execution. [SB10]

instruction types - assignment, input, output, call subroutine, return from subroutine, end of program, and NOP [SB10]. SMS also defines rigid conventions for translating source code to memory, with the first 4 bits of each memory location indicating the op code and the lower 12 bits indicating the operand value, and for setting up stack frames, pushing in order only the return address, parameters, and local variables.

SMS encourages interaction by allowing students to change the contents of instructions in memory and visualize the results on execution, highlighting the instruction in the code pane and the corresponding memory location in the memory pane as the user steps forward and backward in the code. SMS programs, however, are predefined by the instructor and written in the abridged SMS instruction set, conferring no flexibility in changing the high level program during the lab.

SMS's strength lies in its consistency. With its specially designed instruction set, SMS provides automatic idealized stack frame layouts and explicitly draws the parallels between code execution and locations in memory. Its interface, however, provides no visual metaphors for memory. The stack is represented as a memory dump table with no defined separation between abutting stack frames, making the pages of hex digits in the memory pane largely unreadable except for when a specific row is highlighted in relation to its corresponding source line.

With an approach to stack visualization at the mid-level, SMS emphasizes the relationship of code to memory and purges all other CPU state constructs from its execution model. Novice computer architecture students have rated the tool highly on its use in a sample buffer overflow web lab and with its minimal scope, have found the running of the tool relatively self-evident [SB10]. Schweitzer and Boleng plan to incorporate greater flexibility in future iterations such that students can create their own SMS-language programs and assemble them to memory.

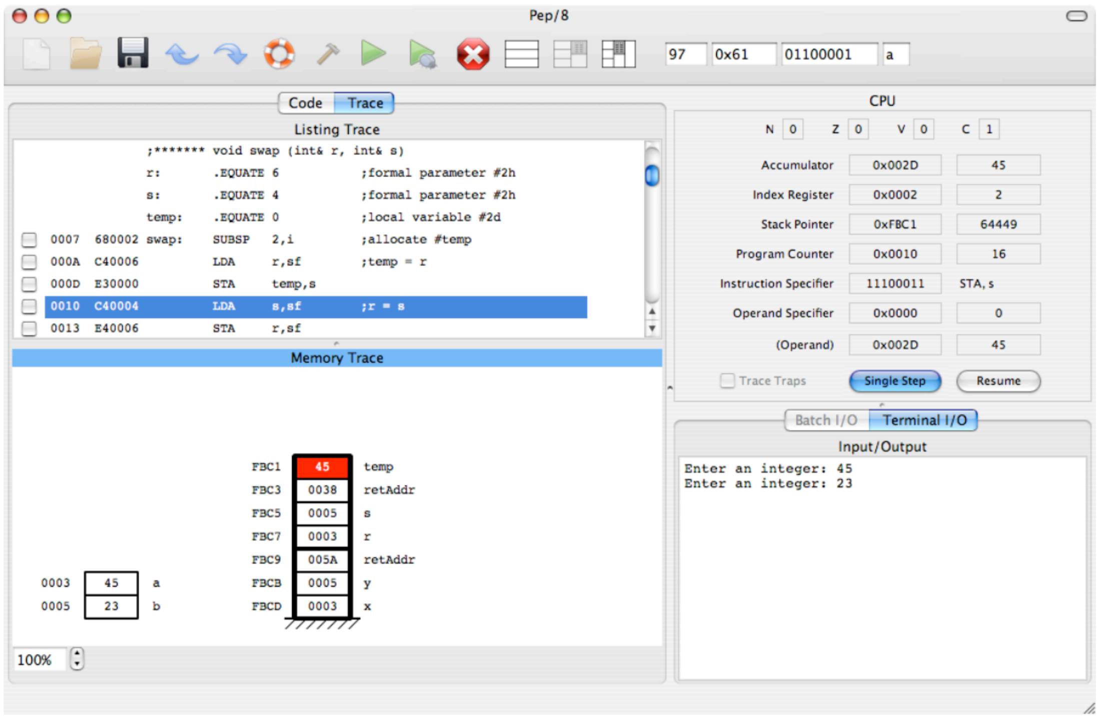
2.4 The Pep/8 Memory Tracer

The Pep/8 Memory Tracer (PMT) is a simulator designed for the Pep/8 virtual machine that features a detailed memory trace facility. The execution model it introduces interprets the machine as a translator between assembly language and machine language with registers maintaining state and a dynamic call stack that explicitly pops and pushes values. Its memory trace pane visualizes the call stack in a way directly parallel to standard figures in textbooks with cells within a frame clearly delineated and labeled by symbol. The consistency with students' existing mental models of the stack discipline makes the tool highly compatible with in-place computer architecture curriculums.

The Pep/8 virtual machine is a small 16-bit von Neumann computer built as a companion to a computer systems textbook for practice with writing machine language and assembly language programs [WD10]. It has a limited instruction and register set with a small subset of instructions mapping directly to changes on the stack. Input programs to PMT must be written in Pep/8-specific assembly, requiring that students learn and understand the new syntax to operate the PMT tool, which can be a burdening cognitive load for students in courses that do not use the Pep/8 companion textbook and for which the language is not familiar.

PMT supports memory tracing through trace tags, which require the user to embed tags in the program comments that indicate how many bytes a symbol takes and how to format and label that memory cell in the trace window [WD10]. In order to achieve a complete visualization, then, the student must specify the number of bytes allocated for each variable and how to display it, making the user responsible for the visualization. This promotes a high level of interaction because the entire assembly program must be documented in this way to avoid trace tag warnings. However, this process can become tedious over time and be a barrier to students who require use of the visualization to facilitate their understanding of the assembly program in the first place.

Once trace tags have been correctly placed, users can step through the Pep/8 assembly program line by line, with the program counter updating to contain the address of the current highlighted instruction. In a separate pane, PMT traces the stack and heap in accordance with the added trace tags. Each stack frame contains its return address and is represented as a separate box with each cell containing its memory address, label, and value, formatted as decimal, hexadecimal, or character as specified by the symbol's trace



```
                ;******* void swap (int& r, int& s)
                r:      .EQUATE 6          ;formal parameter #2h
                s:      .EQUATE 4          ;formal parameter #2h
                temp:   .EQUATE 0          ;local variable #2d
0007   680002   swap:   SUBSP   2,i        ;allocate #temp
000A   C40006           LDA     r,sf       ;temp = r
000D   E30000           STA     temp,s
0010   C40004           LDA     s,sf       ;r = s
0013   E40006           STA     r,sf
```

```
                          FBC1   45     temp
                          FBC3   0038   retAddr
                          FBC5   0005   s
                          FBC7   0003   r
                          FBC9   005A   retAddr
0003   45   a             FBCB   0005   y
0005   23   b             FBCD   0003   x
```

N 0	Z 0	V 0	C 1

Accumulator	0x002D	45
Index Register	0x0002	2
Stack Pointer	0xFBC1	64449
Program Counter	0x0010	16
Instruction Specifier	11100011	STA, s
Operand Specifier	0x0000	0
(Operand)	0x002D	45

```
Enter an integer: 45
Enter an integer: 23
```

Figure 2.3: The red highlighted cell marks the location for the currently executing load instruction in PMT. [WD10]

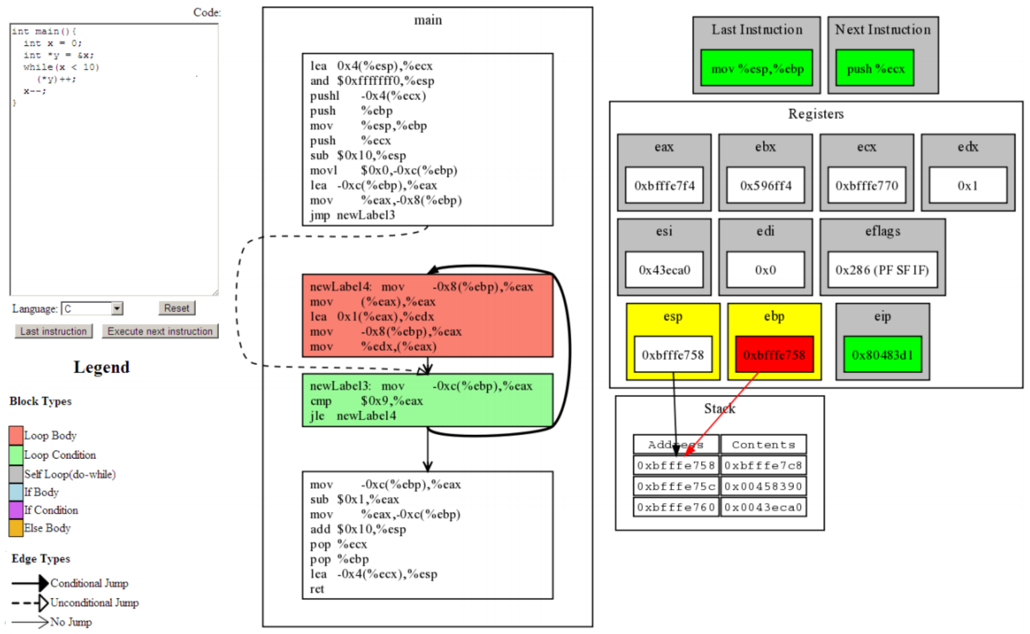
tag. The stack grows and shrinks in line with the program execution, with highlighting on the cell currently being modified.

Resembling the standard stack frame figures, PMT's dynamic call stack visualization presents one of the most easy-to-understand models of the call stack but the constraint that users must insert trace tags that describe the assembly code sets a high bar in terms of accessibility. It also does not provide any mechanism for drawing relationships to higher-level concepts, limited to assembly language and below. The precise memory trace visualizations PMT provides, however, implicitly reinforce the C memory model, displaying the memory structure for globals at a fixed location, locals and parameters on the stack, and dynamic variables on the heap.

2.5 Frances

Frances is a multi-featured visualization tool designed to "help students understand low-level languages, language translation, and computer architecture by showing how familiar high-level code maps to low-level code and how that low-level code behaves on a target architecture" [SPR12]. It leverages students' prior experience with high-level language programming to sketch connections between and explain unfamiliar low-level concepts. Its notional machine considers program execution as a graph of possible execution paths, with blocks of assembly instruction sequences corresponding to standard control structures present in the source code.

Frances contains four main display frames. The first holds a basic text editor that supports user-input C, C++, and FORTRAN programs. The second frame displays the corresponding assembly code, formatted as a control flow graph with blocks containing



```
int main(){
  int x = 0;
  int *y = &x;
  while(x < 10)
    (*y)++;
  x--;
}
```

```
lea  0x4(%esp),%ecx
and  $0xfffffff0,%esp
pushl    -0x4(%ecx)
push     %ebp
mov      %esp,%ebp
push     %ecx
sub  $0x10,%esp
movl     $0x0,-0xc(%ebp)
lea  -0xc(%ebp),%eax
mov      %eax,-0x8(%ebp)
jmp  newLabel3
```

```
newLabel4:  mov      -0x8(%ebp),%eax
mov      (%eax),%eax
lea  0x1(%eax),%edx
mov      -0x8(%ebp),%eax
mov      %edx,(%eax)
```

```
newLabel3:  mov      -0xc(%ebp),%eax
cmp      $0x9,%eax
jle  newLabel4
```

```
mov      -0xc(%ebp),%eax
sub  $0x1,%eax
mov      %eax,-0xc(%ebp)
add  $0x10,%esp
pop  %ecx
pop  %ebp
lea  -0x4(%ecx),%esp
ret
```

Last Instruction: mov %esp,%ebp
Next Instruction: push %ecx

eax	ebx	ecx	edx
0xbfffe7f4	0x596ff4	0xbfffe770	0x1

esi	edi	eflags
0x43eca0	0x0	0x286 (PF SF IF)

esp	ebp	eip
0xbfffe758	0xbfffe758	0x80483d1

Address	Contents
0xbfffe758	0xbfffe7c8
0xbfffe75c	0x00458390
0xbfffe760	0x0043eca0

Figure 2.4: Frances color codes assembly blocks to indicate the type of control structure it corresponds to in the source code. [SPR12]

assembly instructions color-coded to indicate their type - loop body, loop condition, self-loop, if body, if condition, and else body. This explicitly associates less comprehensible assembly syntax with its functionality, defining assembly lines by the role they play in the source program. Links between different blocks are formatted to indicate how the program can move from one to the other, either through straight fall-through, jumps, or conditional jumps.

The third and fourth frames maintain the register and stack state, displaying hexademical values for each of the registers and a hexadecimal address alongside a hexadecimal value for each item on the stack. The stack pointer and base pointer registers direct arrows at the cell on the stack to which they point. For novice programmers, this visualization is non-intuitive as there is no mapping from symbols to their location in registers or on the stack.

In user testing, Frances' designers found that students with a background of 2-3 computer science courses but no computer architecture were able to work through assignments reasonably well with the tool. However, much of the code generation aspects of the visualization were used and better received in more advanced compiler design courses [SPR12]. In its fully-featured state, Frances provides powerful visualizations of low-level to high-level relationships, but the notional machine focuses on a fuller-spectrum of computer architecture than is often necessary for first introductions to the stack discipline.

2.6 Summary

A summary of the four visualization tools with respect to the dimensions of content, accessibility, flexibility, integration, and interaction can be seen in Table 2.1. Each tool interprets program execution under a different lens to emphasize the concepts it considers fundamental to its intended use case. While each applies useful visual metaphors for the call stack, each presents several limitations when considered in the context of novice computer architecture students. In evaluating their design strategies, I seek to build on the most effective components of each in the context of a students first steps into computer architecture, adopting the explicit high- and low- level relationships of Frances, the easy-to-understand standard multi-cell stack frame figures from PMT, the deliberate scope reduction of SMS, and the availability of tooltip detail views in Jype.

Table 2.1: Summary of Program Visualization Tools

Dimension	Jype	SMS	PMT	Frances
Content	High level, Function calls, Variables, Data structures	Mid-level, Source code to memory relationship, Stack as memory dump	Mid-level, Memory trace of assembly code, Dynamic stack, Registers	Low-level, High- to low- level relationship, Assembly control flow graph, Stack as memory dump, Registers
Flexibility	Instructor-defined exercises and examples in Python	Instructor-defined programs in tool-specific language	User-defined programs in tool-specific assembly languge, Must add trace tags to source	User-defined porgrams in C, C++, or FORTRAN
Accessibility	Complete novice programmers, Automatic texutal feedback	Knowledge of basic addressing and memory word interpretation	Some assembly knowledge	Some computer architecture knowledge
Interaction	Step through program, Write code inside tool	Step through program, Can change contents of memory	Step through program, Write code inside tool	Step through program, Write code inside tool
Integration	Textbook-style data structure figures, familiar programming language	Runs on tool-specific memory model	Textbook-style stack frame figures, Runs on tool-specific virtual machine	Familiar programming language, Runs on real architectures

Chapter 3

StackExplorer

3.1 Preface

I propose a stack visualization tool that balances the five dimensions of content, flexibility, accessibility, interaction, and integration to provide early-stage programmers with a tool to facilitate their introduction to basic computer architecture. Leveraging students' prior experience with a high-level programming language, StackExplorer relates familiar high-level constructs to the unfamiliar lower-level concepts through five central elements: a source code pane, an assembly code pane, a call stack pane, a current stack frame pane, and a action toolbar for controlling the execution of the program. At its core, StackExplorer suggests the notional machine of program execution as a series of stack frames that each encloses the symbols accessible in its scope. The tool here limits its scope to concentrate students' attention on the concept of the stack discipline, abstracting away other aspects of a complete execution model.

In the standard use case, the student will load a recursive factorial C program into the tool and step through the programs execution function call by function call. At each call, she will see a new frame pushed onto the call stack and will use the zoom tool to inspect the values in the non-primitive type symbols in the current frame, noting the location and address of the stack pointer and base pointer. After inspecting the assembly instruction at the return address, she will select the previous stack frame and see the matching next-to-execute assembly instruction in that frame's scope. When satisfied, she will run the program to completion, examining the final values of all the variables and the final return value.

In the following sections, the design goals of the tool, its underlying implementation, and intended use cases are discussed.

3.2 Design

StackExplorer's foundation lies in its scoping. It focuses on providing value in regard to student comprehension of the stack discipline, a low-level concept that requires a peripheral understanding of assembly language, memory addressing, hardware registers, and pointers. For many students, this is the first time they see hexadecimal notation, parts of the machine internals, and the strict sequential execution of assembly instructions.

To compensate for the heavy cognitive load of the combination of these new concepts, StackExplorer hides as many details as possible without interfering with the accuracy of its stack discipline model. Studies have shown that the main failure of many visualization tools is that students do not know how to use them effectively, neglecting powerful features because it is unclear when and where each of them should be used [SKM13]. Rather than operate as a generic, do-everything simulator to the point of being overwhelming, StackExplorer offers a small set of possible - but always useful - actions and emphasizes one essential concept: what is on the stack, where is it on stack, and where did it come from? This concept lends itself well to visualization because the call stack is visual by nature.

Integration takes priority as a design goal for several reasons. Previous visualization tools have built off of abstract pedagogical theory to provide incompatible visual metaphors for programming constructs for example, where objects are represented as filing cabinets or constants are represented as engraved stone tablets [SKM13]. While intuitive in a real-world sensibility, these ideas do not translate well to practical programming. Once the student internalizes that an object is a filing cabinet, she will understand the general behavior of objects, but she must then learn how the figures in her textbook represent an object, how the two representations relate, and then associate this with the reality of how objects are actually represented in the machine. While the visual metaphor may help her grasp the concept at first, ultimately she must learn more things down the line by being first presented with the highly theoretical representation.

StackExplorer seeks to maintain consistency with familiar notional machines such that incorporating the tool into an established computer science course does not require an adjustment of the presented execution model. Most computer science students can recall exercises in which they or their professors drew stack frames by hand to illustrate the location of data on the stack, the functionality of the return address, and the movement of the stack pointer. StackExplorer directly mimics these standard drawings of stack frames, with which students are familiar or are becoming familiar, such that the knowledge they gain by working with the tool immediately reinforces what they learn in the classroom. StackExplorer provides an equivalent model, but does it automatically so that students can spend more time exploring and understanding than drawing and struggling.

In addition, StackExplorer ties together the goals of accessibility and flexibility so that introductory students can both own the visualizations but not be required to spend undue time in preparing it. As such, StackExplorer accepts a user-defined program in a pseudo-high level language, C, with which students are already familiar. Content ownership, in contrast to contrived instructor-provided examples, promote greater user engagement because students can experiment, choosing programs that they are interested in visualizing and with which they are comfortable [SKM13]. This also allows for the student to draw connections from the source code instructions they can easily interpret to the lower-level operations they cannot as a description of what is happening at the machine-level and why.

Similar to the ownership notion, StackExplorer runs the program on the user's machine and dynamically obtains the information it visualizes such that the displayed call stack represents the user's actual stack. Thus, the tool is entirely flexible and lends power to the visualization in that it is not an artificially created image but a tangible, albeit abstracted, representation of the users machine state.

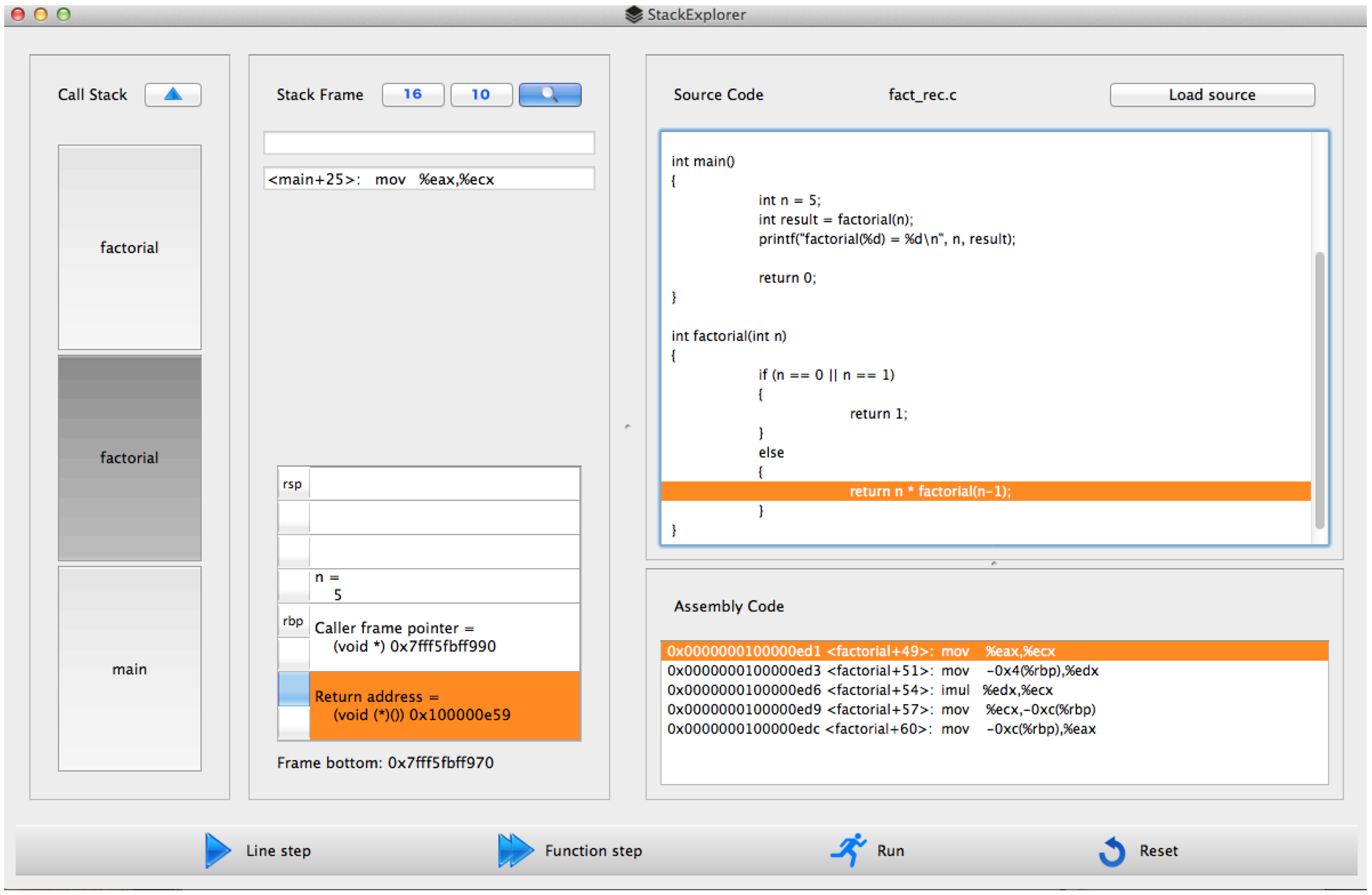
Apart from loading the C program, visualization proceeds automatically. The user does not need to add trace tags, place breakpoints, or provide any other kind of expert user input prior to visualization because novice programmers often do not have a clear sense of what is important in the program and thus have difficulty, for example, in setting useful breakpoints. As a learning tool, there should not be an expectation that the user knows exactly what to do beforehand - the visualization is part of the learning process, not the end goal - so StackExplorer does not ask for any setup of the visualization.

Lastly, as its name suggests, StackExplorer was designed to facilitate exploration. There is no set sequence of actions that a user needs to take during visualization; she can examine the parts in which she is interested and skip through those in which she is not. There are no overwhelming tables of hexadecimal digits or large textual displays. Information is surfaced on demand, either in the form of tooltips for addresses or in the one-line detail view boxes when an element is clicked. The more confusing aspects of the call stack - the fact that it grows downward and is addressed in hex, for example - can be easily righted by flipping the stack so that it looks like it grows upward or by selecting decimal mode. This allows the tool to be true to a real call stack but gives the user the option of simplifying the model first.

In its design, StackExplorer surfaces four main goals in relation to the five evaluation dimensions: simplicity, exploration, ownership, and compatibility with existing notional machines. These primary goals profoundly guided the development of the tool, bringing it to its final format described in the following section.

3.3 Layout

StackExplorer consists of four main panes: source code, assembly code, a call stack, and a selected stack frame. The source code panel displays the currently loaded C program, allowing for the user to select a new source file if desired, automatically compiling the code to a re-locatable location. The control toolbar sits at the bottom of the window with buttons corresponding to line step, function step, run, and reset. Reset sets the currently loaded C program back to its start state to begin execution again, run runs the program to completion from its current state, function step runs the program to the start of its next function call, and line step runs the program to the next source line, stepping into a new function if necessary.



```
<main+25>:   mov   %eax,%ecx
```

```
rsp
n =
  5
rbp  Caller frame pointer =
       (void *) 0x7fff5fbff990
     Return address =
       (void (*)()) 0x100000e59
Frame bottom: 0x7fff5fbff970
```

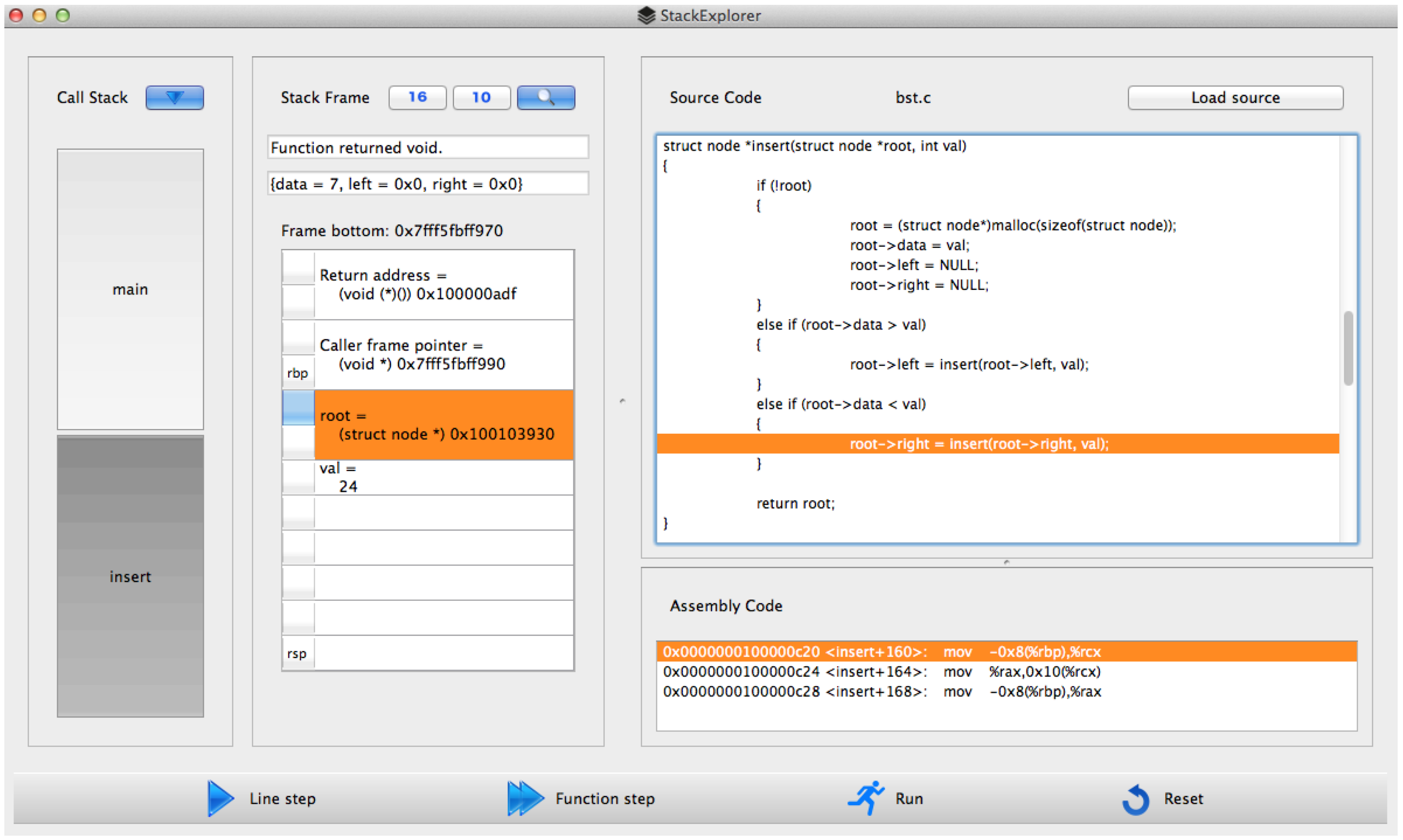
```
int main()
{
        int n = 5;
        int result = factorial(n);
        printf("factorial(%d) = %d\n", n, result);

        return 0;
}

int factorial(int n)
{
        if (n == 0 || n == 1)
        {
                return 1;
        }
        else
        {
                return n * factorial(n-1);
        }
}
```

```
0x0000000100000ed1 <factorial+49>:  mov   %eax,%ecx
0x0000000100000ed3 <factorial+51>:  mov   -0x4(%rbp),%edx
0x0000000100000ed6 <factorial+54>:  imul  %edx,%ecx
0x0000000100000ed9 <factorial+57>:  mov   %ecx,-0xc(%rbp)
0x0000000100000edc <factorial+60>:  mov   -0xc(%rbp),%eax
```

Figure 3.1: A recursive factorial program executing in StackExplorer. In zoom mode, the return address is visualized as the next assembly instruction to execute in the caller's scope. If the user were to select the main stack frame, ¡main+25¿ would be the highlighted isntruction in main's assembly code pane.



```
Function returned void.
{data = 7, left = 0x0, right = 0x0}

Frame bottom: 0x7fff5fbff970
    Return address =
        (void (*)()) 0x100000adf
    Caller frame pointer =
rbp     (void *) 0x7fff5fbff990
    root =
        (struct node *) 0x100103930
    val =
        24
rsp
```

```
struct node *insert(struct node *root, int val)
{
        if (!root)
        {
                root = (struct node*)malloc(sizeof(struct node));
                root->data = val;
                root->left = NULL;
                root->right = NULL;
        }
        else if (root->data > val)
        {
                root->left = insert(root->left, val);
        }
        else if (root->data < val)
        {
                root->right = insert(root->right, val);
        }

        return root;
}
```

```
0x0000000100000c20 <insert+160>:   mov   -0x8(%rbp),%rcx
0x0000000100000c24 <insert+164>:   mov   %rax,0x10(%rcx)
0x0000000100000c28 <insert+168>:   mov   -0x8(%rbp),%rax
```

Figure 3.2: A binary search tree program executing in StackExplorer. The user has just returned from a function and is in zoom mode, inspecting the value of the root node as displayed in the second detail box in the stack frame pane. The stack has also been flipped to grow downward.

All four panels update in response to the toolbar actions. The source code panel highlights the currently executing source line and the assembly code panel displays only the instructions that correspond to the current source line. The call stack panel displays a series of stacked buttons representing the frames currently on the stack. Each frame in this panel is labeled by the name of the function to which it corresponds and can be selected to display its details in the stack frame panel. Selecting a new frame will rewind all other panels to the state of that frame, heavily emphasizing the concept of local scope within each frame. The flip stack button in the upper corner of the panel flips the call stack and stack frame panel so that the stack can be displayed as growing up or down.

The stack frame panel, then, shows the internals of the current frame. It displays the address of the frame base, the return address, the saved base pointer of the caller, the parameters, and the local variables. The symbols are placed in cells, sized to be proportional to the number of bytes they occupy, at their address in the frame, with empty cells left for temporary storage or other unfilled space. Primitive type variables are displayed with their name and formatted value and structs are initially displayed with their type and the address to which they point. StackExplorer detects uninitialized values and displays them as null.

The stack pointer and base pointer are placed next to the cells to which they point and the address of each cell is surfaced in a tooltip on request. There are three mode buttons at the top of the stack frame panel: hex, or default, mode; decimal mode, which converts all the address to decimal values; and zoom mode, which allows for the user to get detailed information about the symbols. In the first two modes, selecting a cell in the stack frame will display the address of the symbol in the second detail view box. In the third mode, selected a cell in the stack frame will display the "zoom value" of the symbol. For primitive type variables, this value is the same but for structs, it will display the names and values of each of its members - abstracting away the heap but still providing the interesting information about the symbol - and for the return address, it will display the assembly instruction to which the return address points.

On a function return, the frame will be removed from the call stack panel and the topmost frame will be automatically selected, with the first detail box in the stack frame pane displaying the value returned from the previous frame.

StackExplorer's interface is condensed to these four core panes to keep the amount of information displayed manageable and the connections between the four panes evident. Any user action updates all four panes and any user action is valid in any state such that there is a minimal learning curve in effectively using the tool.

3.4 Implementation

StackExplorer is wholly dynamic, retrieving all of the information it displays directly from the machine on which it is running. The tool works by running on top of the GNU Debugger (GDB), keeping an open GDB process in the background that maintains the executions current state. Generic commands and scripts have been designed so that they can be applied to any C program and provide an accurate visualization.

When the source code is first loaded, StackExplorer programmatically sets breakpoints at of-interest addresses in the code and starts the GDB process running. When a start-function breakpoint is hit, StackExplorer makes calls to get the current source line number, the disassembled source line, and the values and addresses of the return address, saved base pointer, parameters, local variables, stack pointer, and base pointer. For symbols with zoom values, it makes a separate call to get the value at the address to which those symbols point. It also steps one frame up to update the state of the caller. On each line step, it retrieves and updates the symbols in the current frame and on run, it removes all breakpoints except the one right before the return in main and runs the process to completion. When an end-function breakpoint is hit, StackExplorer finds the return value and jumps to the instruction in the top frames return address.

As a result of being built on top of GDB, StackExplorer is highly generalizable to any machine and architecture and should operate the same. Currently it only supports 32- and 64- bit x86 architectures, but could easily be extended to support any architecture on which GDB runs.

3.5 Summary

StackExplorer was designed to facilitate the conceptual hurdle of moving from high-level programming concepts to the lower-level constructs of the stack discipline. Through a contained stack visualization model, the tool presents a wholly dynamic and compatible view of the call stack of the machine on which it is running. Visualizing the call stack with figures consistent with the notional machines of introductory computer architecture courses, StackExplorer reinforces the models taught in the classroom, offering students an opportunity to freely explore the call stack of a complete, self-defined program.

Chapter 4

Discussion

In holding with its implementation strategy, StackExplorer sacrificed the balance of some of its design goals to remain entirely dynamic in its visualization. Because StackExplorer retrieves all of its data from the machine during execution, it becomes dependent on the compiler and architecture. This is inherent in the design of the tool because it is built on top of GDB, and thus changes cannot be made to break these dependencies without overhauling the entire tool and pursuing a new implementation strategy. This dependency pattern presents several limitations in regard to StackExplorer's use as a learning tool for novice programmers.

StackExplorer is accurate in its visualization of the call stack in relation to GDB, but is entirely reliant on GDB's interpretation of the loaded program, compiled and optimized in debug mode, which is in turn entirely reliant on the architecture on which it is running. While this provides the ownership aspect put forth in the design goals, it makes the visualization subject to unexpected optimizations and other idiosyncrasies of the machine.

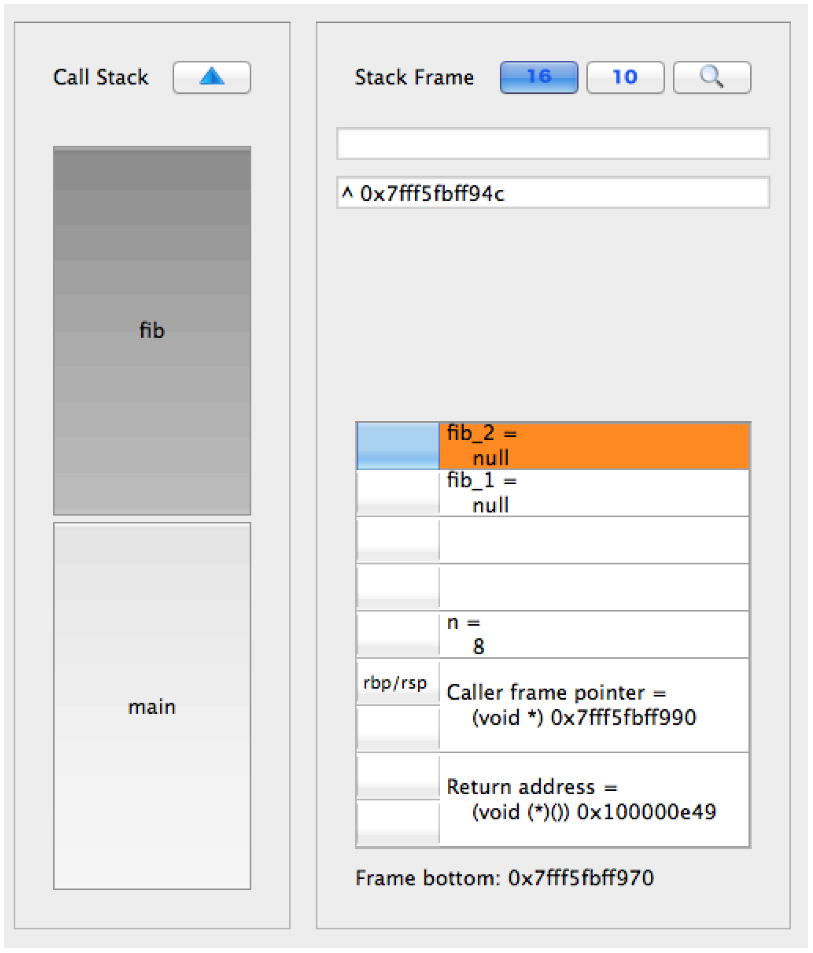
In evaluation of the tool, it is clear that the displayed call stack often does not adhere to the specified calling conventions in the architecture documentation. For example, on a 64-

bit x86 architecture, the first several parameters are stated to be passed in registers rather than on the stack and any parameters passed on the stack are stated to be pushed before the return address [MHJM13]. In the studied sample programs, however, all parameters are pushed onto the stack after the return address as verified by GDB and by manual examination of the corresponding assembly instructions.

In a different regard, there are conventions of specific architectures that the stack does follow that are unexpected. For example, 64-bit x86 architectures reserve a 128-byte space called the red zone on the stack frames of leaf function calls. The red zone is an optimization that guarantees a set scratch area that will not be clobbered by signals or interrupt handlers, and thus the stack pointer does not need to be explicitly moved to store symbols in this space [MHJM13]. An example of this can be seen in Figure x in which the stack pointer, non-intuitively, is at the base of the stack. Similarly, the end of the argument area on the stack must be aligned to 16-bits, often adding several empty cells to the stack.

While these idiosyncrasies can be easily explained by a computer architecture expert, novice programmers will have difficulty understanding the reasons for the different stack layouts and may internalize a special case call stack as the paradigm. In learning the stack discipline for the first time, students prefer idealized stack frames, like those presented in textbooks, where visualizations are consistent among programs and symbols are in a predictable location within each frame. StackExplorer could work to detect and hide these idiosyncrasies but it would sacrifice its dynamic visualization such that the tool would just be constructing stack frames according to a formula.

StackExplorer's other fundamental limitations are its performance, snapshot paradigm, and strict scoping. Because StackExplorer makes several calls to GDB on each user action,



^ 0x7fff5fbff94c

fib_2 =
null
fib_1 =
null
n =
8
rbp/rsp
Caller frame pointer =
(void *) 0x7fff5fbff990
Return address =
(void (*)()) 0x100000e49

Frame bottom: 0x7fff5fbff970

Figure 4.1: A example of the red zone optimization as visualized in StackExplorer. The stack pointer points at the base of the frame and local variables are stored above it.

updates are slow such that users will quickly become frustrated when trying to visualize a multi- function call program. In moving the tool past prototype stage, simple adjustments would have to be made to optimize the GDB commands, make asynchronous calls to the GDB process, or anticipate user actions and preload state.

In terms of the snapshot paradigm employed, StackExplorer visualizes execution as a series of snapshots without the ability to step backward an instruction. Thus there is little sense of history and control flow, which is often useful to the novice programmer. Future versions would benefit from emphasizing the connection between subsequent actions. This could be achieved through the addition of a variety of small features: highlighting both the currently executing and previously executed line, adding a rewind action, color-coding, or highlighting the cells in the stack frame that were updated during the last lines execution.

Limited scoping presents challenges for programs with functions in which no parameters are passed on the stack and no local variables exist. In this case, the stack frame would essentially be empty aside from the return address and saved base pointer, with the interesting functionality occurring in the registers, which are not visualized. A register pane could be added fairly straightforwardly but would risk adding too much content to the tool, which has limited window space.

In its current state, StackExplorer could easily support a range of additional features with minimal code modifications. Before iterating on the current design, however, it is integral to understand how novice programmers interact with the tool and whether they are drawing the intended connections between the panes. The next step in the development process is user testing in which the effectiveness of the tool on learning outcomes can be examined. Through observing usage patterns and collecting feedback, specific features can be classified as useful or not and trimmed, added, or modified accordingly.

Chapter 5

Curriculum Integration

As a learning tool designed under the goal of easy integration with existing curriculums, StackExplorer's essential question is how it can be incorporated into the classroom. Here I consider one example use case for instructors and one for students.

To illustrate recursion at the machine-level, an instructor of an introductory computer science course writes a basic iterative factorial program and a basic recursive factorial program. During lecture, she loads each into StackExplorer simultaneously and steps through each program in conjunction. After several recursive function calls, she zooms in on the return address, displaying the assembly instruction to which it points. She then selects the previous stack frame in the call stack and shows how the instruction in the return address corresponds to the next instruction to execute in the caller, as shown in the assembly code pane. She steps through both programs further until the recursive version starts returning, indicating how the value of the local variable on the stack in the iterative version changes versus how the values are returned to the callers without being stored on the stack in the recursive version. To finish, she runs each to completion to demonstrate that the final return values are equivalent. While simple, this example demonstrates the ubiquity of the call stack throughout computer science curriculums and how StackExplorer can easily facilitate full program traces where previously instructors had to tediously draw analogous diagrams by hand.

As an assignment in an introductory computer architecture course, students are asked to write a C program that recursively reverses a string. The exercise prompts students to make a series of modifications and answer a set of questions. Following are some example questions:

- Examine the memory addresses of the first and second stack frame. Which address is smallest? Which address is largest? What does this suggest about the growth of the stack?
- Zoom in on the return address of the second stack frame. What is its value? Can you find this value anywhere else? Based on your findings, what is the purpose of the return address?
- Change the declaration of your input string from char my_string* to char my_string[10]. How is this array stored? What is the memory address(es) of each?
- Run the program to completion. What instructions are left in the assembly code pane? Look up the definitions of these instructions. What do they do?

Applications of StackExplorer extend beyond the two provided examples. With the tool's visual model of the stack so closely aligned with standard representations and the only input required any basic C program, StackExplorer can be seamlessly integrated into curriculums without dedicating time to setting up the tool or learning how to use and interpret it. Extensions to fit specific contexts can also be easily imagined. For example, adding support for a binary mode and bit ordering, register visualization, instructor hints, text editing, or modifications of values in memory mid-execution could be inserted to adapt the tool to a course's needs.

With the skeleton built out, new modes and front-end components could be attached in under an hour and new data to be visualized could be retrieved simply by introducing a new GDB command and parsing the corresponding output. In this way, StackExplorer is highly extensible to fit instructor's specific needs, whether at a higher or lower level, making it compatible with any curriculum that considers the call stack.

Chapter 6

Conclusions

The call stack appears as a fundamental concept across computer science curriculums, introducing novice programmers to machine-level constructs for the first time. Making the leap from high-level constructs confronts students with the conceptual obstacle of strict accountability for a program's data and memory, upsetting the complacency they develop with languages like Java and Python where they write static words that, by magic, output actions and results.

StackExplorer addresses that knowledge gap by connecting machine- and high- level concepts, highlighting relationships between the source code, assembly code, and call stack. By visualizing the execution model as effects on the call stack and its contained stack frames, StackExplorer emphasizes the separate scoping of stack frames, or function calls, and provides insight into the stack discipline, indicating where symbols are stored in each frame and at what address. Through program visualization, StackExplorer seeks to facilitate student comprehension of the call stack using a simple, accessible, and interactive approach.

Balancing the goals of being dynamic and flexible with being consistent proved more challenging than anticipated, with the tool picking up idiosyncrasies of its host architecture and presenting them to the user. Dynamically retrieving execution information subjects StackExplorer to a series of machine dependencies that lead to non-idealized, unpredictable stack frame structures in certain sample programs. Distributed across a classroom of students running on a variety of different architectures, StackExplorer would give each student a sense of ownership in that she can visualize exactly how her program is executing, but would make it difficult for the instructor to guarantee to comparable user experiences without enforcing that students work on a homogeneous set of laboratory computers. Different architectures would display slightly different stack structures, hindering standardization of the students mental models of program execution.

StackExplorer is a powerful tool for dynamically visualizing the call stack of a program as it runs on the host machine. However, its inconsistencies in the stack discipline as a result of the variations in calling conventions limit its effectiveness as a standalone tool for novice programmers who do not have the expertise to interpret and explain the idiosyncrasies as they arise. In future development, other applications of the tool should be considered with respect to instructor-guided visualizations or architecture-specific call stack instruction. The ultimate step in packaging the tool, then, is pursuing ways in which StackExplorer can be incorporated into introductory computer architecture curriculums as a constructive supplement to the learning process.

Bibliography

[BBE06] Patrick Borunda, Chris Brewer, and Cesim Erten. Gspim: graphical visualization tool for mips assembly programming and simulation. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, pages 244–248, New York, NY, USA, 2006. ACM.

[BK11] Michael David Black and Priyadarshini Komala. A full system x86 simulator for teaching computer organization. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 365–370, New York, NY, USA, 2011. ACM.

[BS10] Jens Bennedsen and Carsten Schulte. Bluej visual debugger for learning the execution of object-oriented programs? *Trans. Comput. Educ.*, 10(2):8:1–8:22, June 2010.

[DCP01] Wanda Dann, Stephen Cooper, and Randy Pausch. Using visualization to teach novices recursion. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 109–112, New York, NY, USA, 2001. ACM.

[Guo13] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[HM10] Juha Helminen and Lauri Malmi. Jype - a program visualization and programming exercise tool for python. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 153–162, New York, NY, USA, 2010. ACM.

[MHJM13] Matz, Hubicka, Jaeger, and Mitchell. System V application binary interface: AMD64 architecure processor supplement, 2013.

[MMSBA04] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 373–376, New York, NY, USA, 2004. ACM.

[Pow04] Kris D. Powers. Teaching computer architecture in introductory computing: Why? and how? In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, pages 255–260, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[PSM$^+$07] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.

[RLKS08] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. Effectiveness of program visualization: A case study with the ville tool. *Journal of Information Technology Education: Innovations in Practice*, 7:IIP 1532, 2008.

[SB10] Dino Schweitzer and Jeff Boleng. A simple machine simulator for teaching stack frames. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 361–365, New York, NY, USA, 2010. ACM.

[SKM13] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*, 13(4):15:1–15:64, November 2013.

[Sor13] Juha Sorva. Notional machines and introductory programming education. *Trans. Comput. Educ.*, 13(2):8:1–8:31, July 2013.

[SPR12] Tyler Sondag, Kian L. Pokorny, and Hridesh Rajan. Frances: A tool for understanding computer architecture and assembly language. *Trans. Comput. Educ.*, 12(4):14:1–14:31, November 2012.

[SS10] Juha Sorva and Teemu Sirkiä. Uuhistle: A software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 49–54, New York, NY, USA, 2010. ACM.

[SS12] Teemu Sirkiä and Juha Sorva. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, pages 19–28, New York, NY, USA, 2012. ACM.

[VS05] Dr. Kenneth Vollmar and Dr. Pete Sanderson. A mips assembly language simulator designed for education. *J. Comput. Sci. Coll.*, 21(1):95–101, October 2005.

[WD10] J. Stanley Warford and Chris Dimpfl. The pep/8 memory tracer: visualizing activation records on the run-time stack. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 371–375, New York, NY, USA, 2010. ACM.

[YYPA01] Cecile Yehezkel, William Yurcik, Murray Pearson, and Dean Armstrong. Three simulator tools for teaching computer architecture: Little man computer, and rtlsim. *J. Educ. Resour. Comput.*, 1(4):60–80, December 2001.