

FORMALIZING THE JAVA VIRTUAL MACHINE IN SEQUENT CALCULUS

by

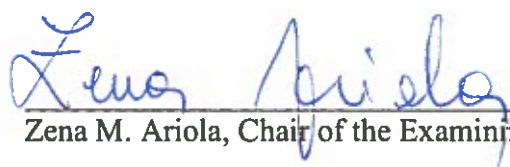
JAMES THOMAS ALLEN

A THESIS

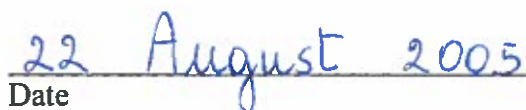
**Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science**

August 2005

“Formalizing the Java Virtual Machine in Sequent Calculus,” a thesis prepared by James Thomas Allen in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:



Zena M. Ariola, Chair of the Examining Committee



Date

Committee in Charge: Professor Zena M. Ariola, Chair
Professor Steven Fickas
Professor Christopher Wilson

Accepted by:



Dean of the Graduate School

© 2005 James Thomas Allen

An Abstract of the Thesis of

James Thomas Allen for the degree of Master of Science
in the Department of Computer and Information Science to be taken August 2005

Title: FORMALIZING THE JAVA VIRTUAL MACHINE IN SEQUENT CALCULUS

Approved:  
Zena M. Ariola

The Java Virtual Machine (JVM) is used extensively in computing environments in which security is a primary concern. The proper formalization of the JVM provides a basis for judgements about the security of JVM programs. Sequent calculi can provide a formalization that allows a shallow encoding of JVM structures into the calculus. This should allow the construction of code verification systems with a smaller trusted code base than the current proof-carrying code technology. This document shows an encoding in Curien and Herbelin calculus, and presents an algorithm for establishing the type-safety of translated programs. Our approach allows a more expressive verification process than traditional methods.

CURRICULUM VITAE

NAME OF AUTHOR: James Thomas Allen

PLACE OF BIRTH: Portland, Oregon

DATE OF BIRTH: May 21, 1963

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon

DEGREES AWARDED:

Master of Science in Computer and Information Science, 2005,
University of Oregon

Bachelor of Science in Computer and Information Science, 2003,
University of Oregon

Bachelor of Science in German, 2003, University of Oregon

AREAS OF SPECIAL INTEREST:

Formal Methods

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science,
University of Oregon, Eugene, 2003-2005

Undergraduate Research Assistant, Department of Computer and Information
Science, University of Oregon, Eugene, 2003-2005

ACKNOWLEDGMENTS

I wish to express appreciation to the members of the committee, Professor Ariola for getting me interested in this topic, Professor Wilson for assistance in preparing this manuscript, and Professor Fickas for providing financial support for me in the Masters' Program.

TABLE OF CONTENTS

Chapter	Page
INTRODUCTION	1
I. ABSTRACT MACHINES	4
I.1 Computing with the Java Virtual Machine	4
I.1.1 JVM Structures	4
I.1.2 The Bytecode Verifier	6
I.1.3. JVM Instruction Semantics	4
I.2. Constructing and Intermediate Abstract Machine	12
I.2.1. The translation from JVMML to the Intermediate Language	12
I.3. Proving the Correctness of the Translation	20
I.4. Incorporating Control Flow Analysis	35
II. A CALCULUS FOR ABSTRACT MACHINES	38
II.1. Developing the Calculus	38
II.1.1. The Calculus of Computational Dualities	38
II.1.2. Substitution at Bounded Depth	40
II.2. Translating Intermediate Code into the Calculus	44
II.2.1. Primitive Values and Contexts, Lists and Arrays	44
II.2.2 Extensions as Functional Continuations	47
II.2.3. Programs and Instructions	48
II.3. Proving the Correctness of the Translation	56
II.3.1. Weakened Normal Forms	56
II.3.2. Confluence of Translated Instructions	61
CONCLUSION	73
BIBLIOGRAPHY	75

LIST OF FIGURES

Figure	Page
I.1. Semantics of several JVMML instructions	9
I.2. Semantics of JVMML invoke and return instructions	10
I.3. PPL instruction set and reduction semantics	15
I.4. Translation from JVMML instructions to PPL instructions	18
II.1. $\lambda\mu\tilde{\mu}$ -calculus grammar	39
II.2. $\lambda\mu\tilde{\mu}$ -calculus reduction semantics.....	40
II.3. $\lambda\mu\tilde{\mu}\hat{\uparrow}$ -calculus grammar	41
II.4. $\lambda\mu\tilde{\mu}\hat{\uparrow}$ -calculus reduction semantics	41
II.5. $\lambda\mu\tilde{\mu}\hat{\uparrow}$ -calculus weakening and substitution rules	42
II.6. Extension function reduction semantics	48
II.7. Translation from PPL to $\lambda\mu\tilde{\mu}\hat{\uparrow}$ -calculus	55

INTRODUCTION

Since its creation, the Java language has become an important language for mobile code. In addition to the normal types of shared libraries that other languages have, Java also provides the applet interface that allows compiled code to travel freely across the Internet. This ability to load and execute pre-compiled code from distant, and often murky sources provides a potential security hole in a host computer system. Because Java code cannot be completely compiled, and still run on any platform, Java must be compiled to an intermediate language which is then interpreted on the host system. Fortunately, this process provides a hook into which the host system can attach security checks.

The Java specification requires that all Java code be compiled into an intermediate language called JVMIL, which is designed to run on an abstract machine called the Java Virtual Machine (JVM). The JVM, then, is an interpreter from the intermediate code to the host platform. This interposition of the JVM allows for dynamic runtime instrumentation that can monitor security constraints. However, no dynamic instrumentation can be assured of success without some guarantees about the type-safety of the object code. In particular, JVMIL partially preserves the type system of Java in the intermediate code. Thus, the JVM is capable of performing a static analysis of the JVMIL code before it is allowed to run. The point of the analysis is to preclude code that subverts the dynamic semantics of the JVM, and thereby bypasses the security instrumentation, or causes the JVM to otherwise malfunction.

The static analysis for the JVM is specified in control flow analytic terms, but vendors are free to perform the analysis in any manner, provided it can guarantee the same

checks as the control flow algorithm. Since the programming languages community is generally more at home with logical analysis, many algorithms have been proposed based on logics of various kinds. Most of these have been logics designed specifically for the JVM. In Stata and Abadi (1999), they develop a logic for JVML0, a subset of 9 JVML instructions, that provides a proof of well-formedness of a program P given a set of initial stack and variable assignments. Freund and Mitchell (1998) extend this to a larger set of JVML instructions, including the instructions dealing with object initialization, and show the soundness of their logic in establishing the well-formedness of programs that do not use uninitialized object references. In Stata and Abadi's approach, the local variables and local stack are assigned types at each program location, and a program type-checks if the constraints can all be satisfied, where the constraints are based on the effect of each instruction on the successor instruction(s).

Qian (1999) takes the approach of assigning to each instruction the least upper bound in the type lattice needed to satisfy the instruction constraints. Again, the logic is highly specific to JVML semantics. In Qian's case, the logic is also highly detailed in its ability to reason about specific instruction widths and branch targets as well as model invoke and return semantics that are missing from Stata and Abadi's work. In contrast, Higuchi and Ohori (2002) assume rudimentary control flow analysis before applying their typing judgements to the JVML. Where the other works assign types to the machine state at program points, Higuchi and Ohori use a sequent style system where the type of a code block is defined in terms of the transformation that it performs to the local state. This is most similar in character to our present work, although the resulting logic is still very specific to the JVM.

In contrast to the work cited above, we hope to create a model for JVM execution that is purely based on an existing calculus. That is, rather than create a calculus that

exactly matches the JVML dynamic semantics, we will show a shallow embedding of the JVML instructions into an existing calculus using the Curry-Howard isomorphism. In chapter I, we formalize the behavior of the JVM in a calculus reminiscent of that used by Stata and Abadi, and then show a translation into an intermediate calculus, presented as an abstract machine semantics. In chapter II, we show the embedding of the intermediate calculus into a derivative of the $\lambda\mu\tilde{\mu}$ calculus of Curien and Herbelin (2000).

CHAPTER 1

ABSTRACT MACHINES

I.1 Computing with the Java Virtual Machine

The Java Virtual Machine has been implemented by various vendors. Although the implementations have changed over time, the JVM Specification itself is only in its second edition [Lindholm and Yellin (1999)]. The specification defines most of the behaviors of the JVM, with some notable exceptions. For example, although automatic garbage collection is generally seen as an integral part of the JVM, the JVM specification leaves the question of garbage collection algorithms up to the implementor, and does not even specify whether the static method areas should be considered part of the garbage collector's purview. Other algorithms, such as the bytecode verifier, are specified in some detail, but the specification allows significant deviation from the stated algorithm provided the outcome is the same as the specification.

I.1.1 JVM Structures

The JVM contains the following structures:

- The Method Area – The method area stores the static structures associated with a class. This stores the code associated with the class methods and also the Runtime Constant Pool consisting of the following entries:
 - Primitive constants used in the program code.
 - Class or interface descriptor, defining the structure and accessibility of the class or interface.

- Method descriptors, defining the structure and accessibility of the methods.
 - Exception handler descriptors, defining the scope and applicability of the exception handlers.
 - Exception handler table, a lookup table of tuples:
{ protected-code-start, protected-code-end, exception-class, handler-entry-point }.
- The Heap – The heap stores all allocated arrays and objects. Each object record contains the object's fields. JVM instructions explicitly allocate space for objects and arrays in the heap. Deallocation is handled through automatic garbage collection, the description of which is outside the JVM spec.
 - The JVM stacks – A JVM stack is allocated for each thread. Each stack contains its own *pc* register that points to the currently executing instruction. A stack frame is created for each method call. Each stack frame consists of the following items:
 - A pointer to the object's runtime constant pool
 - The method's operand stack – The stack size is listed in the method descriptor. The stack is initially empty. The method must neither pop from an empty stack nor push to a full stack.
 - The method's local variable array – The method descriptor gives the size of the local variable array. For non-static methods, the first item is a pointer to the object's heap location. Any other method parameters are loaded into consecutive elements. Any remaining elements are considered to hold the value "undefined" and must not be accessed until they have received values.

The language of JVM instructions is JVMIL. JVMIL instructions can access the local variable array, the stack, and the heap; however, all access to the heap is done indirectly

using an object reference and a method or field name. JVMML instructions cannot access an object's fields or methods without a textual lookup through the class descriptor. JVMML instructions can also affect the pc -register, which points into the current code. A localized JVM state is given by the tuple

$$(\mathcal{C}, pc, \ell, s) \quad (\text{I.1})$$

where \mathcal{C} represents the code to be executed, pc is the pc register, ℓ represents the contents of the local variable array and s represents the local stack. The current method area can be viewed as an array of instructions. The pc -register is a pointer into that array. Thus, the value of the pc is a non-negative number, but that value of an instruction address cannot be confused with a regular int value.

Each JVM stack can be seen as a stack of localized states. Each state holds the current computation in the thread, but only the top-most state is being actively run. To represent the state of the entire machine, we would have to represent the localized state of each thread's JVM stack and the state of the heap. We are not interested in multi-threading, so we assume there is only one thread stack.

The heap can be seen as a map of maps. That is, an object reference is a pointer to a map from method and field names to the code and values they represent.

I.1.2 The Bytecode Verifier

Although most JVMML is compiled by a Java compiler, the JVM does not assume this is true. Since the JVM was designed to acquire code from remote as well as local sources, the code cannot be assumed to be type-safe. Before any code is allowed to execute, the JVM's bytecode verifier must certify that the code passes a minimal set of well-formedness criteria. In particular, the verifier is aware of the required types

of various instructions. For example, the JVM instruction `iadd` requires two integer operands from the stack. Since `iadd` is not defined in the presence of other stack types, the verifier must determine that all control paths produce two integers on top of the stack when the `iadd` is reached.

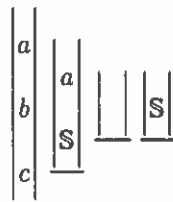
The bytecode verifier is responsible for establishing the following code properties:

- The types of the top values in the stack must match the types expected by the instructions.
- No instruction ever attempts to remove an operand from an empty stack.
- No instruction ever attempts to push a value to a full stack.
- No instruction ever attempts to read or write a local variable outside of the local variable array.
- Each method returns a value of the proper type.
- Each allocated object is initialized before it is used.

In the following section, we combine some of the actions of the verifier with the actions of the bytecode interpreter, since both of these occur at run-time, although the verifier is using a more static set of properties than the interpreter.

1.1.3 JVM Instruction Semantics

The following table lists some useful notation for representing JVM machine states.



stacks of values. We adopt the convention the **S** stands for any number of elements. The first is the stack consisting of a , b and c where only a is accessible. The second is any stack of unknown depth with a at the head. The third is the empty stack. The fourth is any stack of unknown depth, including the empty stack.

- ℓ, \mathcal{E} an array of values
- $|\ell|$ the number of values in ℓ
- $\ell[i]$ for $i \leq |\ell|$, the i^{th} value in ℓ
- $\ell[i..j]$ for $0 < i \leq j \leq |\ell|$, the new array consisting of $\ell[i]$ through $\ell[j]$. In particular, $\ell[i..i]$ is a singleton array.
- $[a, b, c]$ the new array consisting of all of the elements a , b and c , in that order
- $\ell_1 :: \ell_2$ the new array consisting of all of the elements of the array ℓ_1 followed by all the elements of the array ℓ_2
- \boxtimes the only value of the singleton type undefined.
- \square_A a “hole” at the top of the stack waiting to be filled by a value of type A .
- \square_{\perp} a “hole” at the top of the stack that cannot be filled with any value.

if $\mathcal{C}[i] = \text{load } n$ then	$\left(\mathcal{C}, i, \ell, \underline{ \mathbf{S} } \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} \ell[n] \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{store } n$ then	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell[1..n-1]::v::\ell[n+1.. \ell], \underline{ \mathbf{S} } \right)$
if $\mathcal{C}[i] = \text{push } v$ then	$\left(\mathcal{C}, i, \ell, \underline{ \mathbf{S} } \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{pop}$ then	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{ \mathbf{S} } \right)$
if $\mathcal{C}[i] = \text{dup}$ then	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} v \\ v \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{swap}$ then	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v_1 \\ v_2 \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} v_2 \\ v_1 \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{iadd}$ and $v_1 : \text{int}$ and $v_2 : \text{int}$ then	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v_1 \\ v_2 \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} v_1 + v_2 \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{fadd}$ then and $v_1 : \text{float}$ and $v_2 : \text{float}$	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v_1 \\ v_2 \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} v_1 + v_2 \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{goto } a$ then	$\left(\mathcal{C}, i, \ell, \underline{ \mathbf{S} } \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+a, \ell, \underline{ \mathbf{S} } \right)$
if $\mathcal{C}[i] = \text{ifeq } a$ then let $r = \begin{cases} i+a & v = 0 \\ i+1 & v \neq 0 \end{cases}$	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, r, \ell, \underline{ \mathbf{S} } \right)$
if $\mathcal{C}[i] = \text{instanceof } \tau$ then let $r = \begin{cases} 1 & v : \tau \\ 0 & v : \tau' \wedge \tau' \not\prec \tau \end{cases}$	$\left(\mathcal{C}, i, \ell, \underline{\begin{array}{c} v \\ \mathbf{S} \end{array}} \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+1, \ell, \underline{\begin{array}{c} r \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{jsr } a$ then	$\left(\mathcal{C}, i, \ell, \underline{ \mathbf{S} } \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, i+a, \ell, \underline{\begin{array}{c} i+1 \\ \mathbf{S} \end{array}} \right)$
if $\mathcal{C}[i] = \text{ret } n$ then and $\ell[n] : \text{returnAddr}$	$\left(\mathcal{C}, i, \ell, \underline{ \mathbf{S} } \right) \xrightarrow{\text{JVM}} \left(\mathcal{C}, \ell[n], \ell, \underline{ \mathbf{S} } \right)$

Figure I.1: Semantics of several JVM instructions

The localized semantics of a minimal fragment of JVM is given in Figure I.1. Each of those instructions deals only with the local structures, so the transitions just show the change in the top-most JVM stack frame. In particular, the `goto` instruction

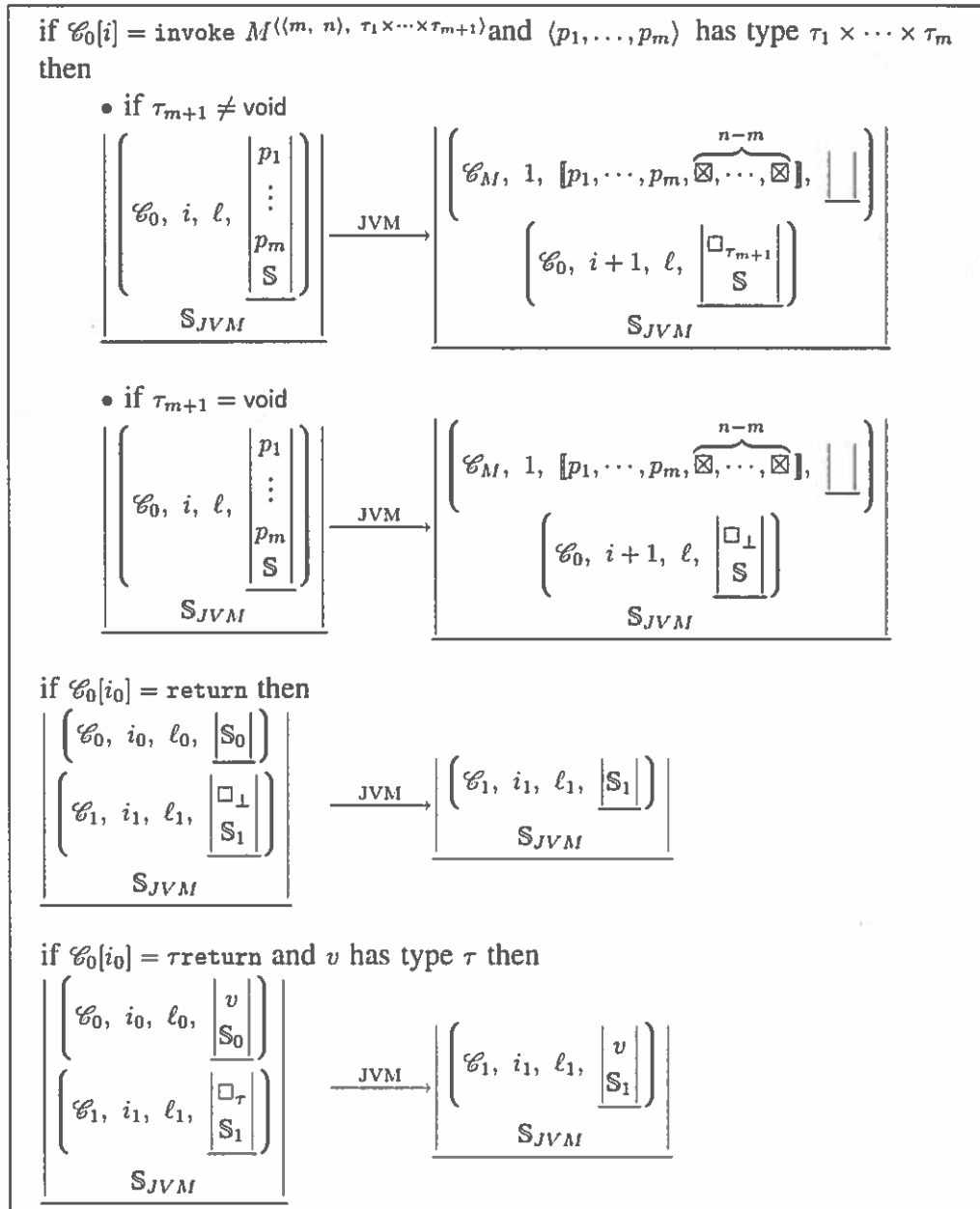


Figure I.2: Semantics of JVMML invoke and return instructions

uses an address offset, which is true to JVMML code, rather than the usual representation as a code label. The invoke and return instructions deal with the JVM stack itself, so they must be specified in terms of the behavior of the whole stack. The semantics of those instructions are given in Figure I.2. Here, the notation $M^{((m, n), T)}$ refers to

a method, and \mathcal{C}_M is the corresponding method code. We assume that M is suitably decorated to find the enclosing object, the details of which we do not model here. The arity of M is indicated by a pair of numbers, $\langle m, n \rangle$ where m indicates the number of parameters that must be passed, and n indicates the total size of the local variable array, including additional space for local results. For any method $M^{\langle(m, n), T\rangle}$, $m \leq n$. Additionally, the signature of M is given as the Cartesian product $\tau_1 \times \dots \times \tau_{m+1}$, indicating that values of types τ_1 through τ_m must appear consecutively on the stack when M is invoked, and the invocation will return with a value of type τ_{m+1} on the stack. The uninhabited type `void` is used to indicate the return type of a method that does not return a value.

The stated semantics for the `invoke` and `return` instructions have taken some liberties with the JVM specification. The specification requires that the bytecode verifier determine statically at load-time that a method returns a value of the correct type. Here, we have replaced this with a run-time comparison on the type of τ_{return} and the type of the hole in the calling stack. Indeed, the very notion of a hole in the calling stack that persists until the `return` instruction is a device of our invention. However, since the point of the current project is to create a system of static checks, the hypothetical dynamic operations really only exist to facilitate the static analysis called for in the JVM specification. In particular, the return value must be double checked in the sense that the type must be checked against the declared type τ of τ_{return} and then checked again against the type of the hole in the caller's stack, which is consistent with the specification.

From the rules in Figure I.2, the JVM does not specify a transition out of the state

$$\left| \left(\mathcal{C}_0, i_0, l_0, \begin{array}{|c|} \hline v \\ \hline \mathbb{S} \\ \hline \end{array} \right) \right|$$

when $\mathcal{C}_0[i_0]$ is `return` or τreturn . Whenever a return is executed on the bottom frame in the JVM stack, the JVM thread of execution enters a finished state. However, to be a valid finished state, if $\mathcal{C}_0[i_0]$ is τreturn , v must be of type τ . The JVM exits when all threads have reached finished states. Of course, to anyone familiar with Java programming, the more common method of terminating the JVM calling the `System.exit()` method. To the JVM static semantics, however, this call appears as a normal method call with a signature `int × void`. The fact that the call never returns is not known to the static semantics.

I.2 Constructing an Intermediate Abstract Machine

To represent the JVM instructions, we will first create an intermediate language that uses only list operations. Since the intermediate language relies mostly on push and pop instructions, we will call it PPL for “Push-Pop Language”. In addition to pushing and popping values, PPL will contain all the instructions needed to simulate JVM.

Internally, the PPL machine uses an array and several lists. The list grammar is as follows:

$$s ::= \odot \mid v \bullet s$$

where v can be a primitive value, the value \boxtimes (undefined), an object reference, or a return address. Lists also have a function that describes their length, defined inductively as follows:

$$|\odot| = 0 \quad |v \bullet s| = |s| + 1$$

To distinguish the type `returnAddr` from numeric values, values of `returnAddr` are denoted “ $@(n)$ ” for some non-negative integer n . The function $\#$ translates between `returnAddr` and integers where $\#(@i) = i$.

The PPL machine contains three list registers, named α , γ , δ . An additional register r holds a value, and the I register holds an instruction queue. An instruction queue is denoted as a semicolon-separated list of instructions, terminated with “!”. The array \mathcal{P} holds an array of instruction queues. As we will see below, each instruction queue can be considered the microcode for a single JVMML instruction. Additionally, the PPL machine has a register Φ that holds the “stored” state. That is, Φ has storage equivalent to each of the structures in the PPL machine, including Φ itself. If Φ is empty, its value is \circ .

The PPL instructions perform the following actions:

const places a primitive constant value into r .

isa if r contains a value of type τ , it is replaced by 1, otherwise it is replaced by 0.

push places the value of r in the front of a list.

pop removes the value at the front of a list and places it in r .

dump clears δ and copies another list into it.

retrieve clears a list and copies δ into it.

dig discards some elements from the front of a list, with the last discarded element placed into r .

transfer removes elements from one list and adds them in reverse order to the front of another list.

call Replaces I with new code and begins execution from the beginning. The new computation begins with an empty α and uses δ in place of γ .

restore Returns the machine to the state prior to the previous call, but executing the instruction following the call.

do causes the numbered instruction queue to be fetched and executed. Optionally, this can choose between instruction queues based on the value of r .

The state of the PPL machine is given in terms of the tuple

$$\left(I, r, \alpha, \gamma, \delta, \langle \mathcal{P}, \Phi \rangle \right)$$

The set of instructions and their reduction semantics are given in Figure I.3. Additionally, the PPL machine has the macros

$$\text{For } n \geq 0, \quad \text{transfer}_n^{k \rightarrow k'} = \{\text{pop}^k; \text{push}^{k'}\}^{(n)}$$

$$\text{dig}_n = \{\text{pop}^\delta\}^{(n)}$$

for any lists k and k' . Here and in succeeding figures for any sequence of symbols S , we let $\{S\}^{(n)}$ represent n consecutive copies of S .

Finally, PPL has one special instruction for type checking. From the description above, PPL largely ignores the types of values in the lists. Specifically, it allows lists to co-mingle elements of all types. The `mustbe` instruction has the following semantics:

$$\left(\text{mustbe}_\tau; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \xrightarrow{\text{PPL}} \left(c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right)$$

if and only if v_0 has type τ .

τ ranges over all the allowable JVM types (except the uninhabited type `void`). If τ contains a value of the correct type, the computation can continue, but if the check fails, then the machine is stuck. Thus, the `mustbe` instruction is something of a super no-op in the sense that either it performs no action and keeps going, or it stops the machine entirely.

$(\text{const } v; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{isa}_\tau; c!, v_0 : \tau, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, 1, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{isa}_\tau; c!, v_0 : \tau', s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, 0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{push}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, v_0 \bullet s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{push}^\gamma; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, v_0 \bullet s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{push}^\delta; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, s_2, v_0 \bullet s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{pop}^\alpha; c!, v_0, v_1 \bullet s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_1, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{pop}^\gamma; c!, v_0, s_1, v_1 \bullet s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_1, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{pop}^\delta; c!, v_0, s_1, s_2, v_1 \bullet s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_1, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{dump}^\odot; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, s_2, \odot, \langle \mathcal{P}, \Phi \rangle)$
$(\text{dump}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, s_2, s_1, \langle \mathcal{P}, \Phi \rangle)$
$(\text{dump}^\gamma; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, s_2, s_2, \langle \mathcal{P}, \Phi \rangle)$
$(\text{retrieve}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_3, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{retrieve}^\gamma; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(c!, v_0, s_1, s_3, s_3, \langle \mathcal{P}, \Phi \rangle)$
$(\text{call}(\mathcal{P}_1); c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(\text{do } @ (1)!, 0, \odot, s_3, \odot, \langle \mathcal{P}_1, (c!, 0, s_1, s_2, \odot, \langle \mathcal{P}_0, \Phi \rangle) \rangle)$
$(\text{restore}!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_1, (c_1!, v_1, s_4, s_5, s_6, \langle \mathcal{P}_2, \Phi \rangle) \rangle)$	$\xrightarrow{\text{PPL}}$	$(c_1!, v_0, s_4, s_5, \odot, \langle \mathcal{P}_2, \Phi \rangle)$
$(\text{do } r!, @ (v_0), s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(\mathcal{P}[v_0], 0, s_1, s_2, \odot, \langle \mathcal{P}, \Phi \rangle)$
$(\text{do } @ (v)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(\mathcal{P}[v], 0, s_1, s_2, \odot, \langle \mathcal{P}, \Phi \rangle)$
for $v_0 = 0$:		
$(\text{do } @ (v_1)?@ (v_2)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(\mathcal{P}[v_1], 0, s_1, s_2, \odot, \langle \mathcal{P}, \Phi \rangle)$
for $v_0 \neq 0$:		
$(\text{do } @ (v_1)?@ (v_2)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle)$	$\xrightarrow{\text{PPL}}$	$(\mathcal{P}[v_2], 0, s_1, s_2, \odot, \langle \mathcal{P}, \Phi \rangle)$

Figure I.3: PPL instruction set and reduction semantics

Example I.2.1 *Starting with the state*

$$\left(\text{do } @ (1)!, 198, 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, 12 \bullet \odot, \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \right)$$

we have the reduction sequence in PPL

$$\begin{array}{l} \left(\begin{array}{l} \text{do } @ (1)!, 198, 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, 12 \bullet \odot, \\ \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, 0, 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, \odot, \\ \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, 0, 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, \\ \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{push}^\alpha; \text{do } @ (2)!, 46, 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, 7 \bullet 3 \bullet \odot, \\ \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{do } @ (2)!, 46, 46 \bullet 1 \bullet \odot, 27 \bullet 46 \bullet 7 \bullet 3 \bullet \odot, 7 \bullet 3 \bullet \odot, \\ \langle \text{dump}^\gamma; \text{dig}_2; \text{push}^\alpha; \text{do } @ (2)!, \odot \rangle \end{array} \right) \end{array}$$

This does not reduce further since $\mathcal{P}[2]$ does not exist.

I.2.1 The translation from JVMML to the Intermediate Language

Example I.2.1 shows how to use δ to access a value from an arbitrary depth in γ and push it onto α . This suggests an encoding of JVMML into PPL using α as the local stack and γ as the local variable array. This leaves δ and r for scratch space. Since δ does not correspond to any JVM structure, we will require that it be cleared after each JVMML instruction. On the other hand, r must always hold something, so we must be cautious about its use in JVMML translations to prevent bleeding information from one instruc-

tion to the next. Figure I.4 shows the translation of the JVMIL instructions into PPL. Here, the value x is not a dynamic reference to the x register of a running JVM, but rather the location in \mathcal{C} of that instruction, a value that is statically available. Thus, the translation only applies to translating an entire code array, and is not well-defined for an individual instruction if its location in its code array is unknown. Also, unlike the uninhabited JVM type `void` in PPL, \perp is a singleton type with \emptyset as the only element. \perp is used as the PPL return type of a JVMIL method that returns `void`.

The translations are faithful to the typing system of JVMIL, but not the dynamic semantics. In particular, the `iadd` and `fadd` instructions check the stack typing and push values of the proper type as results, but do not actually compute the addition, since the PPL machine is not expressive enough to do this. For the remainder of this discussion, we will pretend that `int` values are equal when required to be so, and similarly for `float` values.

In order to define translations from JVM states to PPL states, we have to define the translation $()^\triangleright$ from arrays or stacks of values to lists of values. We do this inductively in the natural way:

$$\begin{array}{l} \text{if } |e| = 0 \text{ then } (e)^\triangleright = \odot \\ \text{otherwise, } (e)^\triangleright = e[1] \bullet (e[2..|e|])^\triangleright \end{array} \left| \begin{array}{l} \left(\begin{array}{|c|} \hline \perp \\ \hline \end{array} \right)^\triangleright = \odot \\ \left(\begin{array}{|c|} \hline v_0 \\ \hline s \\ \hline \end{array} \right)^\triangleright = v_0 \bullet \left(\begin{array}{|c|} \hline s \\ \hline \end{array} \right)^\triangleright \end{array} \right.$$

The translation for the entire JVM stack translates each frame, and must also hold the stack itself in the PPL state. Fortunately, the semantics of Φ permit this. As mentioned above, Φ can be viewed as a linked-list of states. With that in mind, if

$$F_1 = \left(\mathcal{C}, \mathit{x}, \ell, s \right)$$

$(\text{load } i)^\triangleright$	$= \text{dump}^\gamma; \text{dig}_i; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{store } i)^\triangleright$	$= \text{transfer}_{i-1}^{\gamma-\delta}; \text{pop}^\gamma; \text{pop}^\alpha;$ $\text{push}^\gamma; \text{transfer}_{i-1}^{\delta-\gamma}; \text{do}@(\mathit{pc} + 1)!$
$(\text{push } v)^\triangleright$	$= \text{const } v; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{pop})^\triangleright$	$= \text{pop}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{dup})^\triangleright$	$= \text{pop}^\alpha; \text{push}^\alpha; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{swap})^\triangleright$	$= \text{dump}^\alpha; \text{pop}^\alpha; \text{pop}^\alpha; \text{transfer}_2^{\delta-\alpha}; \text{do}@(\mathit{pc} + 1)!$
$(\text{iadd})^\triangleright$	$= \text{pop}^\alpha; \text{mustbe}_{\text{int}}; \text{pop}^\alpha;$ $\text{mustbe}_{\text{int}}; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{fadd})^\triangleright$	$= \text{pop}^\alpha; \text{mustbe}_{\text{float}}; \text{pop}^\alpha;$ $\text{mustbe}_{\text{float}}; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{ifeq } a)^\triangleright$	$= \text{pop}^\alpha; \text{do}(@(\mathit{pc} + a))?\text{do}@(\mathit{pc} + 1)!$
$(\text{instanceof } \tau)^\triangleright$	$= \text{pop}^\alpha; \text{isa}_\tau; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{goto } a)^\triangleright$	$= \text{do}(@(\mathit{pc} + a))!$
$(\text{jsr } a)^\triangleright$	$= \text{const } @(\mathit{pc} + 1); \text{push}^\alpha; \text{do}(@(\mathit{pc} + a))!$
$(\text{ret } n)^\triangleright$	$= \text{dump}^\gamma; \text{dig}_n; \text{do } r!$
$(\text{invoke } M^{(m, n), \tau_1 \times \dots \times \tau_{m+1}})^\triangleright$	$= \text{dump}^\alpha; \{\text{pop}^\alpha; \text{mustbe}_{\tau_i}\}^{(m)} \text{transfer}_m^{\delta-\alpha};$ $\text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \text{transfer}_m^{\alpha-\delta};$ $\text{call}(\mathcal{C}_M)^\triangleright; \text{mustbe}_{\tau_{m+1}}; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!$
$(\text{invoke } M^{(m, n), \tau_1 \times \dots \times \text{void}})^\triangleright$	$= \text{dump}^\alpha; \{\text{pop}^\alpha; \text{mustbe}_{\tau_i}\}^{(m)} \text{transfer}_m^{\delta-\alpha};$ $\text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \text{transfer}_m^{\alpha-\delta};$ $\text{call}(\mathcal{C}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathit{pc} + 1)!$
$(\text{return})^\triangleright$	$= \text{const } \emptyset; \text{restore}!$
$(\tau\text{return})^\triangleright$	$= \text{pop}^\alpha; \text{mustbe}_\tau; \text{restore}!$

Figure I.4: Translation from JVMIL instructions to PPL instructions

then the translation for the JVM stack is

$$\left(\begin{array}{c} F_1 \\ \vdots \\ S \end{array} \right)^\triangleright = \left[(\mathcal{C})^\triangleright [pc], 0, (s)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{c} \boxed{S} \end{array} \right)^\triangleright \right\rangle \right] \quad (I.2)$$

For technical reasons, the translation of the inactive stack frames must be somewhat different, so we use the function $()^\triangleright$, which we define below.

The translation function $()^\triangleright$ creates the following assignments:

$I = (\mathcal{C})^\triangleright [pc]$ The instruction queue holds the translation of instruction to be executed. This simulates the state when the instruction has been fetched, and is about to be decoded and executed.

$r = 0$ The value of r is irrelevant since it has no corresponding JVM structure, provided we can show that the default value 0 is never used in the computation.

$\alpha = (s)^\triangleright$ The local stack is simulated by α .

$\gamma = (\ell)^\triangleright$ The local variable array is simulated by γ .

$\delta = \odot$ δ must be empty since it has no corresponding JVM structure.

$\mathcal{P} = (\mathcal{C})^\triangleright$ The JVM instructions are translated to the equivalent array of instruction queues.

$\Phi = \left(\begin{array}{c} \boxed{S} \end{array} \right)^\triangleright$ The resulting linked list of PPL states is equivalent to the JVM stack of JVM states.

For inactive stack frames, the semantics of `call` and `restore` do not allow the straight forward representation of the instruction queue as $(\mathcal{C})^\triangleright [pc]$. Also, we have not defined a translation for \square_r . Since holes are only placed at the top of a local stack when the frame becomes inactive and are removed when the frame is re-activated, holes can only appear at the top of an inactive stack. Thus, we are not constrained to represent the hole in the local stack, but rather can mark the entire JVM stack frame as somehow

waiting for a value. The translation for holes and x values is given as

$$(x)^\triangleright = \text{do}@(x) \quad (\square_\tau)^\triangleright = \begin{cases} \text{mustbe}_\tau; \text{push}^\alpha & \tau \neq \perp \\ \text{mustbe}_\perp & \tau = \perp \end{cases}$$

Once again, if

$$F_1 = \left(\mathcal{E}, x, \ell, \begin{array}{|c|} \square_\tau \\ \hline S_0 \end{array} \right)$$

then the translation for the JVM stack is

$$\left(\begin{array}{|c|} F_1 \\ \hline S \end{array} \right)^\triangleright = \left((\square_\tau)^\triangleright; (x)^\triangleright!, 0, \left(\begin{array}{|c|} S_0 \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} S \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \quad (\text{I.3})$$

This creates the same assignments as the previous translation, except it uses the instruction queue to indicate both the nature of the hole in the local stack, and the value of x .

1.3 Proving the Correctness of the Translation

Lemma I.3.1 *The translations of all JVM instructions except `invoke`, `return` and `ireturn` preserve the local variable array size.*

Proof. The only instruction that alters γ is `store`. The `store` instruction transfers $i - 1$ elements from γ to δ pops the i^{th} element, pushes it back and transfers the other $1 - 1$ elements back. Thus, the number of elements in γ is unchanged. ■

Lemma I.3.2 *The translations of the JVM instructions `return` and `ireturn` restore the local variable array to the value it held prior to the preceding `invoke`*

Proof. By inspection. No other instructions affect the state stored in Φ . Therefore, this is immediate since both `call` and `restore` leave γ intact. ■

Lemma I.3.3 *No JVM instruction reads from r unless that instruction has first written to it.*

Proof. by inspection. `dig` and `transfer` expand to sequences beginning with `pop`. Thus, all translations except `load`, `swap`, `goto`, `invoke` and `ret` begin with `pop`, `const` or `save`, which write to r . `load`, `swap`, `invoke` and `ret` all perform `dump`, which neither reads nor writes r , followed by `pop`. Thus, they write r before reading it. the translation for `goto` never reads r . ■

Lemma I.3.4 *At the end of each JVM instruction, $\delta = \odot$ and $r = 0$.*

Proof. By inspection. All translations perform either `do` or `restore` before executing the next instruction. `do` sets $\delta = \odot$ and $r = 0$. `restore` returns to the calling instruction queue. Since `invoke` is the only instruction that uses `call`, and `invoke` ends with `do`, `restore` ends by executing `do`. ■

Before proving the main theorem, we must extend the notion of reduction to its reflexive, transitive closure in the usual way. We define $\xrightarrow{\text{VM}}^*$ inductively as

$$S \xrightarrow{\text{VM}} S$$

$$\text{if } S \xrightarrow{\text{VM}} S' \text{ and } S' \xrightarrow{\text{VM}} S'', \text{ then } S \xrightarrow{\text{VM}} S''$$

where “VM” can be either “JVM” or “PPL”. Additionally, we need another relation that simulates a single step of the JVM inside PPL.

Definition I.3.1 We define $\xrightarrow{\text{PPL!}}$ inductively as follows where *do* is any single PPL *do* $r!$, *do* $@(v)!$ or *do* $@(v_1)?@(v_2)!$ instruction and c_j is any arbitrary semicolon-separated sequence of PPL instructions:

$$\begin{array}{l}
 \text{if } \left(do!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL}} \left(c!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \\
 \text{then } \left(do!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL!}} \left(c!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \\
 \\
 \text{if } \left(\text{restore!}, v_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL}} \left(c_1!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_2, \Phi_2 \rangle \right) \\
 \text{and } \left(c_1!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_2, \Phi_2 \rangle \right) \xrightarrow{\text{PPL!}} \left(c_2!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_2, \Phi_2 \rangle \right) \\
 \text{then } \left(\text{restore!}, v_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL!}} \left(c_2!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_2, \Phi_2 \rangle \right) \\
 \\
 \text{if } \left(c_1!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL!}} \left(c_2!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_2, \Phi_2 \rangle \right) \\
 \text{and } \left(c_0; c_1!, v_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL}} \left(c_1!, v_1, \alpha_1, \gamma_1, \delta_1, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \\
 \text{then } \left(c_0; c_1!, v_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_1, \Phi_1 \rangle \right) \xrightarrow{\text{PPL!}} \left(c_2!, v_2, \alpha_2, \gamma_2, \delta_2, \langle \mathcal{P}_1, \Phi_1 \rangle \right)
 \end{array}$$

Clearly, if $S \xrightarrow{\text{PPL!}} S'$, then $S \xrightarrow{\text{PPL}} S'$. The intuition behind $\xrightarrow{\text{PPL!}}$ is that it takes the PPL machine through enough reductions to exhaust the instruction queue once. Since each JVM instruction corresponds to an instruction queue, when applied to JVM translations, this should correspond to a single step of the JVM machine.

Theorem I.3.5 For any valid JVM state S , if $S \xrightarrow{\text{JVM}}^* S'$ then $(S)^\triangleright \xrightarrow{\text{PPL!}}^* (S')^\triangleright$.

Here, the term “valid JVM state” means any final JVM thread state, or any state that can be reached by a finite application of the JVM transition rules given in Figure I.1 and Figure I.2 starting from an initial state $\left[\left[\mathcal{E}_{\text{main}}, 0, \ell, \begin{array}{|c|} \hline \\ \hline \end{array} \right] \right]$ for some program $\mathcal{E}_{\text{main}}$.

Proof. The proof proceeds by cases on the JVM instructions. In each case, we show the reductions in both machines and verify the equality of the translation. In all cases except `invoke` and `return`, it is sufficient to show the top stack frame since the rest of the JVM stack is unchanged in both machines. Thus unless otherwise shown, we let

$$S = \begin{array}{|c|} \hline F \\ \hline \mathbb{S}_J \\ \hline \end{array} \quad \text{and} \quad S' = \begin{array}{|c|} \hline F' \\ \hline \mathbb{S}_J \\ \hline \end{array}.$$

- **Case load:** $F = \left(\mathcal{C}, \mathit{pc}, \ell, \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array} \right)$ where $\mathcal{C}[\mathit{pc}] = \text{load } n$

$$\begin{aligned} F &\xrightarrow{\text{JVM}} \left(\mathcal{C}, \mathit{pc} + 1, \ell, \begin{array}{|c|} \hline \ell[n] \\ \hline \mathbb{S} \\ \hline \end{array} \right) = F' \\ (F)^\triangleright &= \left(\left(\begin{array}{l} \text{dump}^\gamma; \{\text{pop}^\delta; \}^{(n)} \\ \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, \left(\begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left(\left(\begin{array}{l} \{\text{pop}^\delta; \}^{(n)} \text{push}^\alpha; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, \left(\begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, (\ell)^\triangleright, \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!, \ell[n], \left(\begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, (\ell[n+1..|\ell|])^\triangleright, \\ \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\ &\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{do}@(\mathit{pc} + 1)!, \ell[n], \left(\begin{array}{|c|} \hline \ell[n] \\ \hline \mathbb{S} \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, (\ell[n+1..|\ell|])^\triangleright, \\ \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\ &\xrightarrow{\text{PPL}} \left((\mathcal{C})^\triangleright[\mathit{pc} + 1], 0, \left(\begin{array}{|c|} \hline \ell[n] \\ \hline \mathbb{S} \\ \hline \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\ &= (F')^\triangleright \end{aligned}$$

Here, the PPL `dig` command fails if δ is exhausted, but the JVMML command is not defined if $n > |\ell| = |\gamma|$, so this cannot get stuck due to `dig` failure.

- **Case store:** $F = \left(\mathcal{E}, \mathit{x}, \ell, \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array} \right)$ where $\mathcal{E}[\mathit{x}] = \text{store } n$

$$\begin{aligned}
F &\xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{x} + 1, \ell[1..n-1] :: v :: \ell[n+1..|\ell|], \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array} \right) = F' \\
(F)^\triangleright &= \left(\begin{array}{l} \left(\begin{array}{l} \{\text{pop}^\gamma; \text{push}^\delta; \}^{(n-1)} \text{pop}^\gamma; \text{pop}^\alpha; \\ \text{push}^\gamma; \{\text{pop}^\delta; \text{push}^\gamma; \}^{(n-1)} \text{do}@(\mathit{x} + 1)! \end{array} \right), 0, \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\
&\xrightarrow{\text{PPL}} \left(\begin{array}{l} \left(\begin{array}{l} \text{pop}^\gamma; \text{pop}^\alpha; \text{push}^\gamma; \\ \{\text{pop}^\delta; \text{push}^\gamma; \}^{(n-1)} \text{do}@(\mathit{x} + 1)! \end{array} \right), \ell[n-1], \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ (\ell[n..|\ell|])^\triangleright, \ell[n-1] \bullet \dots \bullet \ell[1] \bullet \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\
&\xrightarrow{\text{PPL}} \left(\begin{array}{l} \left(\begin{array}{l} \text{pop}^\alpha; \text{push}^\gamma; \\ \{\text{pop}^\delta; \text{push}^\gamma; \}^{(n-1)} \text{do}@(\mathit{x} + 1)! \end{array} \right), \ell[n], \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ (\ell[n+1..|\ell|])^\triangleright, \ell[n-1] \bullet \dots \bullet \ell[1] \bullet \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\
&\xrightarrow{\text{PPL}} \left(\begin{array}{l} \left(\begin{array}{l} \text{push}^\gamma; \\ \{\text{pop}^\delta; \text{push}^\gamma; \}^{(n-1)} \text{do}@(\mathit{x} + 1)! \end{array} \right), v, \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ (\ell[n+1..|\ell|])^\triangleright, \ell[n-1] \bullet \dots \bullet \ell[1] \bullet \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\
&\xrightarrow{\text{PPL}} \left(\begin{array}{l} \{\text{pop}^\delta; \text{push}^\gamma; \}^{(n-1)} \text{do}@(\mathit{x} + 1)!, v, \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ v \bullet (\ell[n+1..|\ell|])^\triangleright, \ell[n-1] \bullet \dots \bullet \ell[1] \bullet \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right) \\
&\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{do}@(\mathit{x} + 1)!, \ell[1], \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, \\ \ell[1..n-1] \bullet v \bullet (\ell[n+1..|\ell|])^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \end{array} \right)
\end{aligned}$$

$$\begin{aligned} & \xrightarrow{\text{PPL}} \left((\mathcal{C})^\triangleright[x+1], 0, \left(\underline{\mathbf{S}} \right)^\triangleright, \ell[1..n-1] \bullet v \bullet (\ell[n+1..|\ell|])^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right) \\ & = (F')^\triangleright \end{aligned}$$

- **Case push:** $F = \left[\mathcal{C}, \mu, \ell, \left(\underline{\mathbf{S}} \right)^\triangleright \right]$ where $\mathcal{C}[\mu] = \text{push } v$

$$F \xrightarrow{\text{JVM}} \left[\mathcal{C}, \mu+1, \ell, \left(\begin{array}{c} v \\ \underline{\mathbf{S}} \end{array} \right)^\triangleright \right] = F'$$

$$\begin{aligned} (F)^\triangleright & = \left[\text{const } v; \text{push}^\alpha; \text{do}@(\mu+1)!, 0, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right] \\ & \xrightarrow{\text{PPL}} \left[\text{push}^\alpha; \text{do}@(\mu+1)!, v, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right] \\ & \xrightarrow{\text{PPL}} \left[\text{do}@(\mu+1)!, v, v \bullet \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right] \\ & \xrightarrow{\text{PPL}} \left((\mathcal{C})^\triangleright[x+1], 0, v \bullet \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right) \\ & = (F')^\triangleright \end{aligned}$$

- **Case pop:** $F = \left[\mathcal{C}, \mu, \ell, \left(\begin{array}{c} v \\ \underline{\mathbf{S}} \end{array} \right)^\triangleright \right]$ where $\mathcal{C}[\mu] = \text{pop}$

$$F \xrightarrow{\text{JVM}} \left[\mathcal{C}, \mu+1, \ell, \left(\underline{\mathbf{S}} \right)^\triangleright \right] = F'$$

$$\begin{aligned} (F)^\triangleright & = \left[\text{pop}^\alpha; \text{do}@(\mu+1)!, 0, \left(\begin{array}{c} v \\ \underline{\mathbf{S}} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right] \\ & \xrightarrow{\text{PPL}} \left[\text{do}@(\mu+1)!, v, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right] \\ & \xrightarrow{\text{PPL}} \left((\mathcal{C})^\triangleright[x+1], 0, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{C})^\triangleright, \left(\underline{\mathbf{S}_J} \right)^\triangleright \right\rangle \right) \\ & = (F')^\triangleright \end{aligned}$$

• **Case dup:** $F = \left(\mathcal{E}, \mathit{pc}, \ell, \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array} \right)$ where $\mathcal{E}[\mathit{pc}] = \text{dup}$

$$F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{pc} + 1, \ell, \begin{array}{|c|} \hline v \\ \hline v \\ \hline \mathbf{S} \\ \hline \end{array} \right) = F'$$

$$\begin{aligned} (F)^\triangleright &= \left(\text{pop}^\alpha; \text{push}^\alpha; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!, 0, \begin{array}{|c|} \hline v \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left(\text{push}^\alpha; \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!, v, \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left(\text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!, v, v \bullet \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left(\text{do}@(\mathit{pc} + 1)!, v, v \bullet v \bullet \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right) \\ &\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\mathit{pc} + 1], v, v \bullet v \bullet \begin{array}{|c|} \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right) \\ &= (F')^\triangleright \end{aligned}$$

• **Case swap:** $F = \left(\mathcal{E}, \mathit{pc}, \ell, \begin{array}{|c|} \hline v_1 \\ \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array} \right)$ where $\mathcal{E}[\mathit{pc}] = \text{swap}$

$$F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{pc} + 1, \ell, \begin{array}{|c|} \hline v_2 \\ \hline v_1 \\ \hline \mathbf{S} \\ \hline \end{array} \right) = F'$$

$$(F)^\triangleright = \left(\left(\begin{array}{l} \text{dump}^\alpha; \text{pop}^\alpha; \text{pop}^\alpha; \text{pop}^\delta; \\ \text{push}^\alpha; \text{pop}^\delta; \text{push}^\alpha; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, \begin{array}{|c|} \hline v_1 \\ \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \begin{array}{|c|} \hline \mathbf{S}_J \\ \hline \end{array}^\triangleright \right\rangle \right)$$

$$\begin{aligned}
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \left(\begin{array}{l} \text{pop}^\alpha; \text{pop}^\alpha; \text{pop}^\delta; \\ \text{push}^\alpha; \text{pop}^\delta; \text{push}^\alpha; \\ \text{do}@(\mathit{x} + 1)! \end{array} \right), 0, \begin{pmatrix} v_1 \\ v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} v_1 \\ v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \\ \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \left(\begin{array}{l} \text{pop}^\alpha; \text{pop}^\delta; \text{push}^\alpha; \\ \text{pop}^\delta; \text{push}^\alpha; \\ \text{do}@(\mathit{x} + 1)! \end{array} \right), v_1, \begin{pmatrix} v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} v_1 \\ v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \\ \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \left(\begin{array}{l} \text{pop}^\delta; \text{push}^\alpha; \text{pop}^\delta; \\ \text{push}^\alpha; \text{do}@(\mathit{x} + 1)! \end{array} \right), v_2, \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} v_1 \\ v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \\ \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \left(\begin{array}{l} \text{push}^\alpha; \text{pop}^\delta; \\ \text{push}^\alpha; \text{do}@(\mathit{x} + 1)! \end{array} \right), v_1, \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \\ \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \left(\begin{array}{l} \text{pop}^\delta; \text{push}^\alpha; \\ \text{do}@(\mathit{x} + 1)! \end{array} \right), v_1, v_1 \bullet \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} v_2 \\ \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \text{push}^\alpha; \text{do}@(\mathit{x} + 1)!, v_2, v_1 \bullet \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} \text{do}@(\mathit{x} + 1)!, v_2, v_2 \bullet v_1 \bullet \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& \xrightarrow{\text{PPL}} \left[\begin{array}{l} (\mathcal{E})^\triangleright[\mathit{x} + 1], 0, v_2 \bullet v_1 \bullet \begin{pmatrix} \underline{\mathbf{S}} \end{pmatrix}^\triangleright, (\ell)^\triangleright, \odot, \langle (\mathcal{E})^\triangleright, (\underline{\mathbf{S}}_J)^\triangleright \rangle \end{array} \right] \\
& = (F')^\triangleright
\end{aligned}$$

- **Case iadd:** $F = \left(\mathcal{E}, \mathit{pc}, \ell, \begin{array}{|c|} \hline v_1 \\ \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array} \right)$ where $\mathcal{E}[\mathit{pc}] = \text{iadd}$, $v_1 : \text{int}$ and $v_2 : \text{int}$.

$$F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{pc} + 1, \ell, \begin{array}{|c|} \hline v_1 + v_2 \\ \hline \mathbf{S} \\ \hline \end{array} \right) = F'$$

$$(F)^\triangleright = \left(\begin{array}{l} \text{pop}^\alpha; \text{mustbe}_{\text{int}}; \text{pop}^\alpha; \\ \text{mustbe}_{\text{int}}; \text{push}^\alpha; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, \begin{array}{|c|} \hline v_1 \\ \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{mustbe}_{\text{int}}; \text{pop}^\alpha; \\ \text{mustbe}_{\text{int}}; \text{push}^\alpha; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), v_1, \begin{array}{|c|} \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{pop}^\alpha; \text{mustbe}_{\text{int}}; \\ \text{push}^\alpha; \text{do}@(\mathit{pc} + 1)! \end{array} \right), v_1, \begin{array}{|c|} \hline v_2 \\ \hline \mathbf{S} \\ \hline \end{array}^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{mustbe}_{\text{int}}; \text{push}^\alpha; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), v_2, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\text{push}^\alpha; \text{do}@(\mathit{pc} + 1)!, v_2, \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\text{do}@(\mathit{pc} + 1)!, v_2, v_2 \bullet \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\mathit{pc} + 1], 0, v_2 \bullet \left(\underline{\mathbf{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbf{S}}_J \right)^\triangleright \right\rangle \right)$$

$$= (F')^\triangleright$$

- **Case fadd:** This case is similar to the case for iadd.

- **Case goto:** $F = \left(\mathcal{E}, \mathit{pc}, \ell, \underline{\mathbf{S}} \right)$ where $\mathcal{E}[\mathit{pc}] = \text{goto } a$

$$F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{pc} + a, \ell, \underline{\mathbf{S}} \right) = F'$$

$$\begin{aligned}
(F)^\triangleright &= \left(\text{do@}(\underline{x} + a)!, 0, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\underline{x} + a], 0, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&= (F')^\triangleright
\end{aligned}$$

• **Case ifeq:** $F = \left(\mathcal{E}, \underline{x}, \ell, \left(\begin{array}{c} v \\ \underline{S} \end{array} \right) \right)$ where $\mathcal{E}[\underline{x}] = \text{goto } a$

• **Case $v = 0$:** $F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \underline{x} + a, \ell, \underline{S} \right) = F'$

$$\begin{aligned}
(F)^\triangleright &= \left(\text{pop}^a; \text{do@}(\underline{x} + a)?\text{do@}(\underline{x} + 1)!, 0, \left(\begin{array}{c} v \\ \underline{S} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \right. \\
&\quad \left. \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&\xrightarrow{\text{PPL}} \left(\text{do@}(\underline{x} + a)?\text{do@}(\underline{x} + 1)!, v, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\underline{x} + a], 0, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&= (F')^\triangleright
\end{aligned}$$

• **Case $v \neq 0$:** $F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \underline{x} + a, \ell, \underline{S} \right) = F'$

$$\begin{aligned}
(F)^\triangleright &= \left(\text{pop}^a; \text{do@}(\underline{x} + a)?\text{do@}(\underline{x} + 1)!, 0, \left(\begin{array}{c} v \\ \underline{S} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \right. \\
&\quad \left. \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&\xrightarrow{\text{PPL}} \left(\text{do@}(\underline{x} + a)?\text{do@}(\underline{x} + 1)!, v, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\underline{x} + 1], 0, \left(\underline{S} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{S}_J \right)^\triangleright \right\rangle \right) \\
&= (F')^\triangleright
\end{aligned}$$

• **Case instanceof:** $F = \left(\mathcal{E}, \underline{x}, \ell, \left(\begin{array}{c} v \\ \underline{S} \end{array} \right) \right)$ where $\mathcal{E}[\underline{x}] = \text{instanceof } \tau$

• Case v has type τ : $F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{x}+1, \ell, \begin{array}{|c|} \hline 1 \\ \hline \mathbb{S} \\ \hline \end{array} \right) = F'$

$$\begin{aligned}
 (F)^\triangleright &= \left(\text{pop}^\alpha; \text{isa}_\tau; \text{push}^\alpha; \text{do}@(\mathit{x}+1)!, 0, \begin{array}{|c|} \hline v \\ \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{isa}_\tau; \text{push}^\alpha; \text{do}@(\mathit{x}+1)!, v, \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{push}^\alpha; \text{do}@(\mathit{x}+1)!, 1, \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{do}@(\mathit{x}+1)!, 1, 1 \bullet \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\mathit{x}+1], 0, 1 \bullet \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &= (F')^\triangleright
 \end{aligned}$$

• Case v does not have type τ : $F \xrightarrow{\text{JVM}} \left(\mathcal{E}, \mathit{x}+1, \ell, \begin{array}{|c|} \hline 0 \\ \hline \mathbb{S} \\ \hline \end{array} \right) = F'$

$$\begin{aligned}
 (F)^\triangleright &= \left(\text{pop}^\alpha; \text{isa}_\tau; \text{push}^\alpha; \text{do}@(\mathit{x}+1)!, 0, \begin{array}{|c|} \hline v \\ \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{isa}_\tau; \text{push}^\alpha; \text{do}@(\mathit{x}+1)!, v, \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{push}^\alpha; \text{do}@(\mathit{x}+1)!, 0, \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left(\text{do}@(\mathit{x}+1)!, 0, 0 \bullet \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &\xrightarrow{\text{PPL}} \left((\mathcal{E})^\triangleright[\mathit{x}+1], 0, 0 \bullet \begin{array}{|c|} \hline \mathbb{S} \\ \hline \end{array}, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{|c|} \hline \mathbb{S}_J \\ \hline \end{array} \right)^\triangleright \right\rangle \right) \\
 &= (F')^\triangleright
 \end{aligned}$$

- **Case jsr:** $F = \left[\mathcal{E}, \mathit{pc}, \ell, \underline{\mathbb{S}} \right]$ where $\mathcal{E}[\mathit{pc}] = \text{jsr } a$

$$F \xrightarrow{\text{JVM}} \left[\mathcal{E}, \mathit{pc} + a, \ell, \begin{array}{c} i+1 \\ \underline{\mathbb{S}} \end{array} \right] = F'$$

$$\begin{aligned} (F)^\triangleright &= \left[\text{const } @(\mathit{pc} + 1); \text{push}^\alpha; \text{do}@(\mathit{pc} + a)!, 0, \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[\text{push}^\alpha; \text{do}@(\mathit{pc} + a)!, @(\mathit{pc} + 1), \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[\text{do}@(\mathit{pc} + a)!, @(\mathit{pc} + 1), @(\mathit{pc} + 1) \bullet \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[(\mathcal{E})^\triangleright[\mathit{pc} + a], 0, @(\mathit{pc} + 1) \bullet \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &= (F')^\triangleright \end{aligned}$$

- **Case ret:** $F = \left[\mathcal{E}, \mathit{pc}, \ell, \underline{\mathbb{S}} \right]$ where $\mathcal{E}[\mathit{pc}] = \text{ret } n$

$$F \xrightarrow{\text{JVM}} \left[\mathcal{E}, \ell[n], \ell, \underline{\mathbb{S}} \right] = F'$$

$$\begin{aligned} (F)^\triangleright &= \left[\text{dump}^\gamma; \{\text{pop}^\delta; \}^{(n)}; \text{do } r!, 0, \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[\{\text{pop}^\delta; \}^{(n)}; \text{do } r!, 0, \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \ell[1] \bullet \dots \bullet |\ell|, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[\text{do } r!, \ell[n], \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \ell[n+1] \bullet \dots \bullet |\ell|, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &\xrightarrow{\text{PPL}} \left[(\mathcal{E})^\triangleright[\#(\ell[n])], \ell[n], \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \ell[n+1] \bullet \dots \bullet |\ell|, \left\langle (\mathcal{E})^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right] \\ &= (F')^\triangleright \end{aligned}$$

In this case, $\#(\ell[n])$ is only defined if $\ell[n] = @(\mathit{i})$ for some i , which is also required for the $\xrightarrow{\text{JVM}}$ reduction.

- Case invoke with void:

$$\begin{aligned}
 S &= \left[\begin{array}{c} \mathcal{E}_0, \mathit{pc}, \ell, \\ \left[\begin{array}{c} p_1 \\ \vdots \\ p_m \\ \mathcal{S} \end{array} \right] \\ \mathcal{S}_J \end{array} \right] \quad \text{where } \mathcal{E}_0[\mathit{pc}] = \text{invoke } M^{((m, n), \tau_1 \times \dots \times \tau_m \times \text{void})} \\
 S &\xrightarrow{\text{JVM}} \left[\begin{array}{c} \left(\mathcal{E}_M, 1, [p_1, \dots, p_m, \overbrace{\square, \dots, \square}^{n-m}], \left[\begin{array}{c} \square \\ \square \end{array} \right] \right) \\ \left(\mathcal{E}_0, \mathit{pc} + 1, \ell, \left[\begin{array}{c} \mathcal{S} \end{array} \right] \right) \\ \mathcal{S}_J \end{array} \right] = S' \\
 (S')^\triangleright &= \left[\begin{array}{c} (\mathcal{E}_M)^\triangleright[1], 0, \left(\left[\begin{array}{c} \square \\ \square \end{array} \right] \right)^\triangleright, \left([p_1, \dots, p_m, \overbrace{\square, \dots, \square}^{n-m}] \right)^\triangleright, \odot, \\ \left\langle (\mathcal{E}_M)^\triangleright, \left(\left[\begin{array}{c} \mathcal{E}_0, \mathit{pc} + 1, \ell, \left[\begin{array}{c} \mathcal{S} \end{array} \right] \end{array} \right] \right)^\triangleright \right\rangle \\ \mathcal{S}_J \end{array} \right] \\
 &= \left[\begin{array}{c} (\mathcal{E}_M)^\triangleright[1], 0, \odot, p_1 \bullet \dots \bullet p_m \bullet \overbrace{\square \bullet \dots \bullet \square}^{n-m} \bullet \odot, \odot, \\ \left\langle (\mathcal{E}_M)^\triangleright, \left((\square_\perp)^\triangleright; (\mathit{pc} + 1)^\triangleright!, 0, (\ell)^\triangleright, \left(\left[\begin{array}{c} \mathcal{S} \end{array} \right] \right)^\triangleright, \odot, \left\langle (\mathcal{E}_0)^\triangleright, \left(\left[\begin{array}{c} \mathcal{S}_J \end{array} \right] \right)^\triangleright \right\rangle \right) \right\rangle \end{array} \right] \\
 &= \left[\begin{array}{c} (\mathcal{E}_M)^\triangleright[1], 0, \odot, p_1 \bullet \dots \bullet p_m \bullet \overbrace{\square \bullet \dots \bullet \square}^{n-m} \bullet \odot, \odot, \\ \left\langle (\mathcal{E}_M)^\triangleright, \left(\left(\begin{array}{c} \text{mustbe}_\perp; \\ \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, (\ell)^\triangleright, \left(\left[\begin{array}{c} \mathcal{S} \end{array} \right] \right)^\triangleright, \odot, \left\langle (\mathcal{E}_0)^\triangleright, \left(\left[\begin{array}{c} \mathcal{S}_J \end{array} \right] \right)^\triangleright \right\rangle \right) \right\rangle \end{array} \right] \\
 (S)^\triangleright &= \left[\begin{array}{c} \left(\begin{array}{c} \text{dump}^\alpha; \{\text{pop}^\alpha; \text{mustbe}_{\tau_i}\}^{(m)} \\ \text{transfer}_m^{\delta \rightarrow \alpha}; \text{dump}^\odot; \\ \text{const } \square; \{\text{push}^\delta; \}^{(n-m)} \\ \text{transfer}_m^{\alpha \rightarrow \delta}; \text{call}(\mathcal{E}_M)^\triangleright; \\ \text{mustbe}_\perp; \text{do}@(\mathit{pc} + 1)! \end{array} \right), 0, \left(\begin{array}{c} p_1 \\ \vdots \\ p_m \\ \mathcal{S} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \left\langle (\mathcal{E}_0)^\triangleright, \left(\left[\begin{array}{c} \mathcal{S}_J \end{array} \right] \right)^\triangleright \right\rangle \end{array} \right]
 \end{aligned}$$

$$\begin{aligned}
&= \left[\begin{array}{l} \left(\begin{array}{l} \text{dump}^\alpha; \{\text{pop}^\alpha; \text{mustbe}_{\tau_i}\}^{(m)} \text{transfer}_m^{\delta \rightarrow \alpha}; \text{dump}^\ominus; \text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \\ \text{transfer}_m^{\alpha \rightarrow \delta}; \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)! \\ 0, p_1 \dots p_m \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \ominus, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \{\text{pop}^\alpha; \text{mustbe}_{\tau_i}\}^{(m)} \text{transfer}_m^{\delta \rightarrow \alpha}; \text{dump}^\ominus; \text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \\ \text{transfer}_m^{\alpha \rightarrow \delta}; \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)! \\ 0, p_1 \dots p_m \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, p_1 \dots p_m \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{transfer}_m^{\delta \rightarrow \alpha}; \text{dump}^\ominus; \text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \\ \text{transfer}_m^{\alpha \rightarrow \delta}; \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)! \\ p_m, (\mathbb{S})^\triangleright, (\ell)^\triangleright, p_1 \dots p_m \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{dump}^\ominus; \text{const } \boxtimes; \{\text{push}^\delta; \}^{(n-m)} \text{transfer}_m^{\alpha \rightarrow \delta}; \\ \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)! \\ p_m, p_m \dots p_1 \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \left(\underline{\mathbb{S}} \right)^\triangleright, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{transfer}_m^{\alpha \rightarrow \delta}; \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)!, \\ \boxtimes, p_m \dots p_1 \cdot \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, \overbrace{\boxtimes \dots \boxtimes}^{n-m} \cdot \ominus, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)!, \\ \boxtimes, \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, p_1 \dots p_m \cdot \overbrace{\boxtimes \dots \boxtimes}^{n-m} \cdot \ominus, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{call}(\mathcal{E}_M)^\triangleright; \text{mustbe}_\perp; \text{do}@(\mathcal{I}+1)!, \\ \boxtimes, \left(\underline{\mathbb{S}} \right)^\triangleright, (\ell)^\triangleright, p_1 \dots p_m \cdot \overbrace{\boxtimes \dots \boxtimes}^{n-m} \cdot \ominus, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \end{array} \right) \\ \xrightarrow{\text{PPL}} \left(\begin{array}{l} \text{do } @(\mathcal{V})!, 0, \ominus, p_1 \dots p_m \cdot \overbrace{\boxtimes \dots \boxtimes}^{n-m} \cdot \ominus, \ominus, \\ \left\langle (\mathcal{E}_M)^\triangleright, \left(\begin{array}{l} \text{mustbe}_\perp; \\ \text{do}@(\mathcal{I}+1)! \end{array} \right), 0, (\ell)^\triangleright, \left(\underline{\mathbb{S}} \right)^\triangleright, \ominus, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathbb{S}}_J \right)^\triangleright \right\rangle \right\rangle \end{array} \right) \end{array} \right]
\end{aligned}$$

$$\begin{aligned}
 & \xrightarrow{\text{PPL}} \left((\mathcal{E}_M)^\triangleright [1], 0, \odot, p_1 \bullet \dots \bullet p_m \bullet \overbrace{\boxtimes \bullet \dots \bullet \boxtimes}^{n-m} \bullet \odot, \odot, \right. \\
 & \left. \left\langle (\mathcal{E}_M)^\triangleright, \left(\left(\begin{array}{c} \text{mustbe}_\perp; \\ \text{do}@(\mathcal{P}+1)! \end{array} \right), 0, (\ell)^\triangleright, \left(\underline{\mathcal{S}} \right)^\triangleright, \odot, \left\langle (\mathcal{E}_0)^\triangleright, \left(\underline{\mathcal{S}_J} \right)^\triangleright \right\rangle \right\rangle \right\rangle \right) \\
 & = (S')^\triangleright
 \end{aligned}$$

This only succeeds in either machine if $p_i : \tau_i$ for all $0 < i \leq m$. In the JVM, the result is undefined otherwise. In the PPL machine, the `mustbe` reductions won't occur, and the machine will get stuck.

- Case `invoke` with non-void:

$$S = \left(\begin{array}{c} \left(\begin{array}{c} p_1 \\ \vdots \\ p_m \\ \underline{\mathcal{S}} \end{array} \right) \\ \mathcal{E}_0, \mathcal{P}, \ell, \\ \underline{\mathcal{S}_J} \end{array} \right) \quad \begin{array}{l} \text{where } \mathcal{E}_0[\mathcal{P}] = \text{invoke } M^{((m, n), \tau_1 \times \dots \times \tau_m \times \tau_{m+1})} \\ \text{and } \tau_{m+1} \neq \text{void} \end{array}$$

This is similar to the previous case.

- Case `τ return`:

$$S = \left(\begin{array}{c} \left(\begin{array}{c} v \\ \underline{\mathcal{S}_0} \end{array} \right) \\ \mathcal{E}_0, \mathcal{P}_0, \ell_0, \\ \left(\begin{array}{c} \square_\tau \\ \underline{\mathcal{S}_1} \end{array} \right) \\ \underline{\mathcal{S}_J} \end{array} \right) \quad \text{where } \mathcal{E}_0[\mathcal{P}] = \tau\text{return} \text{ and } v : \tau$$

$$S \xrightarrow{\text{JVM}} \left(\begin{array}{c} \left(\begin{array}{c} v \\ \underline{\mathcal{S}_1} \end{array} \right) \\ \mathcal{E}_1, \mathcal{P}_1, \ell_1, \\ \underline{\mathcal{S}_J} \end{array} \right) = S'$$

$$\begin{aligned}
(S)^\triangleright &= \left[\begin{array}{c} \text{pop}^\alpha; \text{mustbe}_\tau; \text{restore}!, 0, \left(\begin{array}{c} v \\ \underline{S_0} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \\ \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{c} \mathcal{E}_1, \mathit{pc}_1, \ell_1, \left(\begin{array}{c} \square_\tau \\ \underline{S_1} \end{array} \right)^\triangleright \\ \underline{S_J} \end{array} \right) \right\rangle \end{array} \right] \\
&= \left[\begin{array}{c} \text{pop}^\alpha; \text{mustbe}_\tau; \text{restore}!, 0, \left(\begin{array}{c} v \\ \underline{S_0} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \\ \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{c} (\square_\tau)^\triangleright; (\mathit{pc}_1)^\triangleright!, 0, \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right\rangle \end{array} \right) \right] \\
&\xrightarrow{\text{PPL}} \left[\begin{array}{c} \text{restore}!, v, \left(\begin{array}{c} \underline{S_0} \end{array} \right)^\triangleright, (\ell)^\triangleright, \odot, \\ \left\langle (\mathcal{E})^\triangleright, \left(\begin{array}{c} (\square_\tau)^\triangleright; (\mathit{pc}_1)^\triangleright!, 0, \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right\rangle \right\rangle \end{array} \right] \\
&\xrightarrow{\text{PPL}} \left[\begin{array}{c} (\square_\tau)^\triangleright; (\mathit{pc}_1)^\triangleright!, v, \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right] \\
&= \left[\begin{array}{c} \left(\begin{array}{c} \text{mustbe}_\tau; \\ \text{push}^\alpha; \\ \text{do}@(\mathit{pc}_1)! \end{array} \right), v, \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right] \\
&\xrightarrow{\text{PPL}} \left[\begin{array}{c} \text{do}@(\mathit{pc}_1)!, v, v \bullet \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right] \\
&\xrightarrow{\text{PPL}} \left[\begin{array}{c} (\mathcal{E}_1)^\triangleright[\mathit{pc}_1]!, 0, v \bullet \left(\begin{array}{c} \underline{S_1} \end{array} \right)^\triangleright, (\ell_1)^\triangleright, \odot, \left\langle (\mathcal{E}_1)^\triangleright, \left(\begin{array}{c} \underline{S_J} \end{array} \right)^\triangleright \right\rangle \right] = (S')^\triangleright
\end{array} \right]
\end{aligned}$$

- **Case return:** This is similar to the previous case.

■

I.4 Incorporating Control Flow Analysis

The PPL machine does not have anything like the JVM's pc register; so, in PPL, the flow of control from one instruction to the next must be made explicit. This provides

an opportunity in translation to optimize the PPL instruction queues to eliminate many of the `do` instructions by simply appending an instruction to the preceding one. For example, the translation of

$$\llbracket \dots, \text{push } 2, \text{store } 3, \dots \rrbracket$$

if `push 2` occurs in the i^{th} position in \mathcal{P} is

$$\text{const } 2; \text{push}^\alpha; \text{do}@ (i + 1)! \bullet \text{dump}; \text{transfer}_2^{\gamma \rightarrow \delta}; \text{pop}^\gamma; \text{pop}^\alpha; \text{push}^\gamma; \text{transfer}_2^{\delta \rightarrow \gamma}; \text{do}@ (i + 2)!$$

but we could eliminate the internal control flow statement and just translate it as the single entry in \mathcal{P} :

$$\text{const } 2; \text{push}^\alpha; \text{dump}; \text{transfer}_2^{\gamma \rightarrow \delta}; \text{pop}^\gamma; \text{pop}^\alpha; \text{push}^\gamma; \text{transfer}_2^{\delta \rightarrow \gamma}; \text{do}@ (i + 1)!$$

In the optimized code, the last instruction is `do@(i + 1)!` rather than the original `do@(i + 2)!`, because one instruction queue has been lost. Thus, the transformed code no longer has the property $\mathcal{P}[i] = (\mathcal{E}[i])^\triangleright$. In performing the transformation, we must be careful to maintain the same code entry points as the original translation. That is, if the JVM program has an instruction `goto + 2`, then we must maintain the `do` instruction after the following instruction queue so that the `+2` branch target still exists.

The algorithm for optimizing the PPL code is simple:

1. Consider each instruction address i starting from $|\mathcal{P}|$ down to 1.
2. If the only reference to $@(i)$ is `do @(i)!` or `do @(j)?@(i)!` in the instruction at $\mathcal{P}[i - 1]$, then
3. replace `do @(i)!` with `dump⊙`;
4. for every $j > i$, replace every occurrence in \mathcal{P} of $@(j)$ with $@(j - 1)$.

This algorithm works because the PPL machine marks `returnAddr` types in such a way that even in the instruction `const @(5)`, the address 5 can be easily discerned. The semantics of a program that has undergone this transformation are unchanged. In particular branching to labels that are beyond the size of \mathcal{P} in the unmodified program will cause the modified program to get stuck at the same point. Although the new PPL program does not clear r as frequently, Lemma I.3.3 says that this is unnecessary.

The algorithm, as stated, requires $O(n^3)$, but we can alter it to make one pass that simply records all of the code label references. Then, in a second pass through the code, it can compute the label offset at each point, replace the needed labels with their offset labels, and remove the unnecessary labels. Applying this algorithm to the translation is equivalent to Higuchi and Ohori's approach of reconstructing code labels by identifying branch targets and then assigning labels. While their explanation is more traditional in control flow analysis, neither approach seems to have an advantage in complexity. We have adopted our strategy because its use is orthogonal to the question of the typing system.

CHAPTER II

A CALCULUS FOR ABSTRACT MACHINES

II.1 Developing the Calculus

II.1.1 The Calculus of Computational Dualities

The $\lambda\mu\tilde{\mu}$ -calculus serves as a starting place for modeling the PPL machine since it has successfully been shown to model other abstract machines (Bohannon (2004)). The calculus was first discovered in Curien and Herbelin (2000). The syntax for the $\lambda\mu\tilde{\mu}$ -calculus is given in Figure II.1. The symbol x stands for any regular variable, usually written as lower case roman letters, whereas, the symbol k stands for any context variable, usually written as lowercase Greek letters.

The basic computational unit is the command, which associates a term with a context. The most familiar form of a command is $\langle v \mid \tilde{\mu}x.c \rangle$. This brings together a context on the right with a term on the left. The context contains holes, represented as free occurrences of the variable x in c , which are filled by v .

For example, if $c = \langle v \mid \tilde{\mu}x.2 + x \rangle$ and v reduces to 1, then $\langle v \mid \tilde{\mu}x.c \rangle$ should reduce to $2 + 1$. Thus, the left side of the command produces a value or term, which is consumed by the continuation on the right. Dually, the form $\langle \mu k.c \mid e \rangle$ executes c with all free occurrences of the continuation variable k bound to e . Of course, the form $\langle \mu k.c \mid \tilde{\mu}x.c \rangle$ can also occur, and the priority of the reductions in this case distinguishes a call-by-name versus call-by-value strategy (Curien and Herbelin (2000)). Since the JVM uses call-by-value strategy, we adopt the convention that priority be given to μ .

As in λ -calculus, a lambda abstraction binds a regular variable to the first term in the argument list. That is, the \cdot (cons) operator constructs a list of terms terminated

c	$::=$	$\langle v \mid e \rangle$
v	$::=$	$x \mid \mu k.c \mid \lambda x.v \mid e \cdot v$
e	$::=$	$k \mid \tilde{\mu}x.c \mid v \cdot e \mid k\lambda.e$

Figure II.1: $\lambda\mu\tilde{\mu}$ -calculus grammar

by a context, and the λ operator deconstructs it. Thus in a command, regular lambda abstraction only operates on a term list. The form $\langle \lambda x.v \mid v' \cdot e \rangle$ reduces to $\langle v \mid [x \setminus v'] \mid e \rangle$. Here we use “ $v \mid [x \setminus v']$ ” to indicate that free occurrences of x are replaced by v' in v . As usual, the binding for x only extends to the body of the lambda abstraction and does not occur on the other side of the command.

The dual to λx is $k\lambda$, context lambda abstraction. Predictably, the form $\langle e' \cdot v \mid k\lambda.e \rangle$ reduces to $\langle v \mid e \mid [k \setminus e'] \rangle$. Thus, $k\lambda$ deconstructs lists of contexts that are terminated by a term in exactly the same way that regular lambda abstraction deconstructs term lists that are terminated by a context. This mixing of terms and contexts in lists can cause confusion in the exhibited list forms. For example, in a complicated list such as

$$e_1 = 1 \cdot 2 \cdot \epsilon \quad e_2 = 3 \cdot \epsilon \quad v_1 = e_1 \cdot e_2 \cdot \epsilon \cdot \epsilon \quad e_3 = v_1 \cdot 4 \cdot \epsilon$$

the proper nesting of lists in e_3 is given as

$$((1 \cdot 2 \cdot \epsilon) \cdot (3 \cdot \epsilon) \cdot \epsilon \cdot \epsilon) \cdot 4 \cdot \epsilon$$

which is not ambiguous if one has access to the parse tree. Therefore, the term $\lambda x.\mu k.c$ must produce $c \mid [x \setminus 1 \cdot 2 \cdot \epsilon \cdot 3 \cdot \epsilon \cdot \epsilon \cdot \epsilon] \mid [k \setminus 4 \cdot \epsilon]$ rather than $c \mid [x \setminus 1] \mid [k \setminus 2 \cdot \epsilon \cdot 3 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 4 \cdot \epsilon]$. The fact that this is not obvious from the form $1 \cdot 2 \cdot \epsilon \cdot 3 \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot 4 \cdot \epsilon$ should be viewed as typographical omission of parentheses that are actually inserted in the formal representation.

(β_-)	$\langle \lambda x.v \mid v' \cdot e \rangle$	\rightarrow	$\langle v [x \setminus v'] \mid e \rangle$
(β_-)	$\langle e' \cdot v \mid k \lambda.e \rangle$	\rightarrow	$\langle e [k \setminus e'] \mid v \rangle$
(μ)	$\langle \mu k.c \mid e \rangle$	\rightarrow	$c [k \setminus e]$
$(\tilde{\mu})$	$\langle v \mid \tilde{\mu} x.c \rangle$	\rightarrow	$c [x \setminus v]$

Figure II.2: $\lambda\mu\tilde{\mu}$ -calculus reduction semantics

II.1.2 Substitution at Bounded Depth

The semantics of the $\lambda\mu\tilde{\mu}$ -calculus involves the notion of binding through substitution. For example, we read $c [x \setminus v]$ as the command that results from replacing all free occurrences of x with v . However, finding all of the free occurrences of a variable involves an unbounded search through the syntax tree of the c . Instead of this primitive notion of substitution, we will extend the $\lambda\mu\tilde{\mu}$ -calculus with immediate substitution. That is, we will reify the bindings into the term calculus and resolve them when they reach the top of the syntax tree. This new calculus is referred to as the $\lambda\mu\tilde{\mu}\uparrow$ -calculus. In order to introduce the binding operator $[x \leftarrow v]$, we must also introduce its dual, the weakening operator \uparrow^x . While the binding operator introduces a substitution, the weakening operator eliminates one. The weakening operator is necessary to avoid unbounded search for possible redexes through lists of bindings, as can happen in explicit substitution calculi. The syntax for the $\lambda\mu\tilde{\mu}\uparrow$ -calculus is given in Figure II.3. In addition to support for immediate substitution, we have two constant symbols, ϵ and ε . ϵ stands for the null context. That is, a context that cannot supply anything to a term. Similarly, ε stands for the “undefined” term.

As we encounter substitutions at the top of the syntax tree, we will move them inward in the terms. Therefore, all of the productions of the $\lambda\mu\tilde{\mu}\uparrow$ syntax must accommodate the new u production. For example, when a command undergoes substitution,

u	$::=$	$[k \leftarrow e] \mid [x \leftarrow v]$
w	$::=$	$\uparrow^x \mid \uparrow^k$
W	$::=$	$w \mid Ww$
c	$::=$	$\langle v \mid e \rangle \mid c W \mid c u$
v	$::=$	$x \mid \mu k.c \mid \lambda x.v \mid e \cdot v \mid \exists \mid v W \mid v u$
e	$::=$	$k \mid \bar{\mu}x.c \mid v \cdot e \mid k\lambda.e \mid \epsilon \mid e W \mid e u$

Figure II.3: $\lambda\mu\bar{\mu}\uparrow$ -calculus grammar

(μ)	$\langle \mu k.c \mid e \rangle$	\rightarrow	$c[k \leftarrow e]$
$(\bar{\mu})$	$\langle v \mid \bar{\mu}x.c \rangle$	\rightarrow	$c[x \leftarrow v]$
(β_-)	$\langle \lambda x.v \mid v' \cdot e \rangle$	\rightarrow	$\langle v' \mid \bar{\mu}x.\langle v \mid e \uparrow^x \rangle \rangle$
(β_-)	$\langle e' \cdot v \mid k\lambda.e \rangle$	\rightarrow	$\langle \mu k.\langle v \uparrow^k \mid e \rangle \mid e' \rangle$

Figure II.4: $\lambda\mu\bar{\mu}\uparrow$ -calculus reduction semantics

the substitution is simply distributed through both parts.

$$\langle v \mid e \rangle [r \leftarrow 0] \text{ reduces to } \langle v[r \leftarrow 0] \mid e[r \leftarrow 0] \rangle$$

Wherever a substitution meets a variable, we require that either the substitution be made immediately, or the variable be weakened sufficiently to prevent the substitution. Thus, we have the rules $x[x \leftarrow v] \rightarrow v$ and $k \uparrow^x [x \leftarrow v] \rightarrow k$ but, for example, $k[x \leftarrow v]$ does not reduce.

In our form of the calculus, we must be very careful about handling λ abstractions to maintain the priority of resolving the terms before contexts (call-by-value). In our modified calculus, we follow Herbelin and Curien and simulate the $\lambda\mu\bar{\mu}$ lambda abstractions using the $\bar{\mu}$ construction with the appropriate weakenings. The modified semantics is shown in Figure II.4.

In addition to the modifications for call-by-value, we must also include enough rules to deal effectively with substitutions and weakenings. The remaining reduction semantics are given in Figure II.5. Here, the notation \uparrow^V represents a series of consecutive

Weakening Introduction		
$(\lambda\tau)$	$(\lambda x.v) [k \leftarrow e]$	$\rightarrow \lambda x.(v [k \leftarrow e \uparrow^x])$
$(\tau\lambda e)$	$(k\lambda.v) [k' \leftarrow e]$	$\rightarrow \lambda x.(v [k' \leftarrow e \uparrow^k])$
$(\tau\lambda v)$	$(k\lambda.v) [x \leftarrow v]$	$\rightarrow \lambda x.(v [x \leftarrow v \uparrow^k])$
$(\bar{\mu}\tau)$	$(\bar{\mu}x.c) [k \leftarrow e]$	$\rightarrow \bar{\mu}x.(c [k \leftarrow e \uparrow^x])$
$(\mu\tau e)$	$(\mu k.c) [k' \leftarrow e]$	$\rightarrow \mu k.(c [k' \leftarrow e \uparrow^k])$
$(\mu\tau v)$	$(\mu k.c) [x \leftarrow v]$	$\rightarrow \mu k.(c [x \leftarrow v \uparrow^k])$
Substitution Elimination		
$(x\tau)$	$x [x \leftarrow v]$	$\rightarrow x$
$(k\tau)$	$k [k \leftarrow e]$	$\rightarrow k$
(τv)	$\uparrow^{V \cup x} [x \leftarrow v]$	$\rightarrow \uparrow^V$
(τe)	$\uparrow^{V \cup k} [k \leftarrow e]$	$\rightarrow \uparrow^V$
(sv)	$\mu k. \langle v \uparrow^k \mid k \rangle$	$\rightarrow v$
(se)	$\bar{\mu}x. \langle x \mid e \uparrow^x \rangle$	$\rightarrow e$
Substitution Distributive Laws		
$(\cdot\tau)$	$(v \cdot e) u$	$\rightarrow v u \cdot e u$
$(\tau\cdot)$	$(e \cdot v) u$	$\rightarrow e u \cdot v u$
$(c\tau)$	$\langle v \mid e \rangle u$	$\rightarrow \langle v u \mid e u \rangle$
Weakening Factorization Laws		
$(c \uparrow^x)$	$c \uparrow^{V \cup x} [k \leftarrow e \uparrow^{V' \cup x}]$	$\rightarrow c \uparrow^V [k \leftarrow e \uparrow^{V'}] \uparrow^x$
$(c \uparrow^k e)$	$c \uparrow^{V \cup k} [k' \leftarrow e \uparrow^{V' \cup k}]$	$\rightarrow c \uparrow^V [k' \leftarrow e \uparrow^{V'}] \uparrow^k$
$(c \uparrow^k v)$	$c \uparrow^{V \cup k} [x \leftarrow e \uparrow^{V' \cup k}]$	$\rightarrow c \uparrow^V [x \leftarrow e \uparrow^{V'}] \uparrow^k$
$(v \uparrow^x)$	$v \uparrow^{V \cup x} [k \leftarrow e \uparrow^{V' \cup x}]$	$\rightarrow v \uparrow^V [k \leftarrow e \uparrow^{V'}] \uparrow^x$
$(v \uparrow^k e)$	$v \uparrow^{V \cup k} [k' \leftarrow e \uparrow^{V' \cup k}]$	$\rightarrow v \uparrow^V [k' \leftarrow e \uparrow^{V'}] \uparrow^k$
$(v \uparrow^k v)$	$v \uparrow^{V \cup k} [x \leftarrow v' \uparrow^{V' \cup k}]$	$\rightarrow v \uparrow^V [x \leftarrow v' \uparrow^{V'}] \uparrow^k$
$(e \uparrow^x)$	$e \uparrow^{V \cup x} [k \leftarrow e' \uparrow^{V' \cup x}]$	$\rightarrow e \uparrow^V [k \leftarrow e' \uparrow^{V'}] \uparrow^x$
$(e \uparrow^k e)$	$e \uparrow^{V \cup k} [k' \leftarrow e' \uparrow^{V' \cup k}]$	$\rightarrow e \uparrow^V [k' \leftarrow e' \uparrow^{V'}] \uparrow^k$
$(e \uparrow^k v)$	$e \uparrow^{V \cup k} [x \leftarrow v \uparrow^{V' \cup k}]$	$\rightarrow e \uparrow^V [x \leftarrow v \uparrow^{V'}] \uparrow^k$

Figure II.5: $\lambda\mu\bar{\mu}\uparrow$ -calculus weakening and substitution rules

substitutions that include every variable in a set V of regular and context variables. In fact, we use set notation to refer to a contiguous group of weakenings because the rules do not distinguish the order of weakenings in such a case. Thus, the W production is considered a single entity where the redex search is concerned. However, since the number of weakenings in a single W is limited to the number of variables in the formula, having a defined set of variables will make this a constant search depth. The important property is that redexes occur at bounded depth from the top of the tree.

The motivation behind a rule such as

$$(\mu k.c)[x \leftarrow v] \rightarrow \mu k.(c[x \leftarrow v \uparrow^k])$$

is to ensure that v , which was formed outside of the μ abstraction, not be subject to the binding for k created by the abstraction. That is, if v contains an occurrences of k , then it must already be bound to something before the effect of the μ abstraction. This property allows us to have “immediate” substitution, even though we only actually make the replacement when the substitution is at the top of the syntax tree. Thus, for each abstraction, we may have to move n substitutions through it, but n varies with the number of substitution variables, not the parse tree of the term being reduced, and for any given abstract machine, the number of substitution variables is constant. To assist with working with weakenings in representing PPL constructs, we adopt the following abbreviations:

$$\begin{array}{lll} \uparrow^* & \triangleq & \uparrow^{\varphi} \uparrow^{\gamma} \uparrow^{\alpha} \uparrow^s \uparrow^r \uparrow^{\delta} \\ \uparrow^{-\varphi} & \triangleq & \uparrow^{\gamma} \uparrow^{\alpha} \uparrow^s \uparrow^r \uparrow^{\delta} \\ \uparrow^{-\gamma} & \triangleq & \uparrow^{\varphi} \uparrow^{\alpha} \uparrow^s \uparrow^r \uparrow^{\delta} \\ \uparrow^{-\alpha} & \triangleq & \uparrow^{\varphi} \uparrow^{\gamma} \uparrow^s \uparrow^r \uparrow^{\delta} \\ \uparrow^{-s} & \triangleq & \uparrow^{\varphi} \uparrow^{\gamma} \uparrow^{\alpha} \uparrow^r \uparrow^{\delta} \\ \uparrow^{-r} & \triangleq & \uparrow^{\varphi} \uparrow^{\gamma} \uparrow^{\alpha} \uparrow^s \uparrow^{\delta} \\ \uparrow^{-\delta} & \triangleq & \uparrow^{\varphi} \uparrow^{\gamma} \uparrow^{\alpha} \uparrow^s \uparrow^r \end{array}$$

Also, we adopt the usage of an anonymous variable “_” to represent a variable that is bound and then immediately weakened. For example, we have the reduction

$$\langle \lambda_ . \mu k . c \mid v \cdot e \rangle \rightarrow c[k \leftarrow e]$$

Here, the anonymous variable construction can be seen as syntactic sugar for the construction

$$\langle \lambda y . ((\mu k . c) \uparrow^y) \mid v \cdot e \rangle$$

where y is a variable not otherwise used in the computation. Since anonymous variable bindings do not persist, they do not affect the complexity of the redex search.

II.2 Translating Intermediate Code into the Calculus

II.2.1 Primitive Values and Contexts, Lists and Arrays

To work with the JVM, we hypothesize the existence of primitive values and their dual, primitive contexts. The primitive values are the usual set of scalar integer and floating point numbers. The primitive contexts have the form $[]_{int}$ and $[]_{float}$. A command such as $\langle 0 \mid []_{int} \rangle$ is finished. It is not necessarily stuck in the usual sense, because it may occur in a side computation. That is, it may occur as a form in the syntax tree that will never reach the top of the tree, so the dynamic semantics will never attempt to reduce it. In fact, we intend to use primitive contexts in just that manner. They will be temporarily bound in a way that the static type checker requires them to be well-typed without realizing that they do not contribute to the computation of the program.

For lists, we have the following formal translation from PPL list forms to $\lambda\mu\tilde{\mu}\uparrow$ contexts:

$$\begin{aligned} (v)^\circ &= v && \text{for primitive and reference types} \\ (v \bullet l)^\circ &= (v)^\circ \cdot (l)^\circ \end{aligned}$$

The PPL machine also maintains some arrays, which $\lambda\mu\tilde{\mu}\uparrow$ does not support. To handle this situation, we will simply translate the array to a PPL list and then translate the list:

$$(\ell)^\circ = ((\ell)^\triangleright)^\circ$$

In PPL, we had direct access to array elements without processing the entire list, but in $\lambda\mu\tilde{\mu}\uparrow$, we seem to have lost that ability when we translated arrays into lists. Thus, if ℓ is the list of “instructions”, and the term M represents the current state of the computation. The instruction `do @(i)!` translates to a command with the following inductive definition:

$$\begin{aligned} \text{do } @(\mathit{i})! &= \langle \ell \uparrow^* \mid A_{\mathit{i}} \rangle \\ A_1 &= \sigma\lambda.\tilde{\mu}-. \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \\ A_{\mathit{i}} &= \sigma\lambda. (A_{\mathit{i}-1} \uparrow^\sigma) \end{aligned}$$

This peels off i contexts, retaining only the last one in σ , puts the remaining list in the anonymous variable, and supplies M to the i^{th} context. Assuming the environment contains substitutions for all the PPL structures, but no current binding for ℓ , then the reduction sequence looks something like the following:

$$\begin{aligned} &\langle (\ell)^\circ \uparrow^* \mid \{\sigma\lambda.(\cdot)^{(i-1)} \sigma\lambda.\tilde{\mu}-. \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-1)}\} \rangle [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots}] [\alpha \leftarrow e_2 \uparrow^{\cdots}] \cdots \\ \rightarrow^* &\langle (\ell)^\circ \mid \sigma\lambda.(\{\sigma\lambda.(\cdot)^{(i-2)} \sigma\lambda.\tilde{\mu}-. \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-2)}\} \uparrow^\sigma [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots) \rangle \\ \rightarrow^* &\left\langle \begin{array}{l} \mu\sigma. \langle (\ell[2..|\ell|])^\circ \uparrow^\sigma \mid (\{\sigma\lambda.(\cdot)^{(i-2)} \sigma\lambda.\tilde{\mu}-. \langle M \uparrow^\sigma \\ \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-2)} \uparrow^\sigma [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots) \rangle \mid \ell[1] \end{array} \right\rangle \end{aligned}$$

$$\begin{aligned}
& \rightarrow^* \left\langle \begin{array}{l} (\ell[2..|\ell|])^\circ \mid \sigma\lambda.(\{\sigma\lambda.\}^{(i-3)} \sigma\lambda.\tilde{\mu}_-. \langle M \uparrow^\sigma \\ \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-3)} \uparrow^\sigma \uparrow^\sigma [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots [\sigma \leftarrow \ell[1] \uparrow^\sigma]) \end{array} \right\rangle \\
& \quad \vdots \\
& \rightarrow^* \left\langle \begin{array}{l} (\ell[i..|\ell|])^\circ \mid \sigma\lambda.(\tilde{\mu}_-. \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-1)} \\ [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots [\sigma \leftarrow \ell[1] \{\uparrow^\sigma\}^{(i-1)}] \cdots [\sigma \leftarrow \ell[i-1] \uparrow^\sigma]) \end{array} \right\rangle \\
& \rightarrow^* \left\langle \begin{array}{l} (\ell[i+1..|\ell|])^\circ \mid \tilde{\mu}_-. \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-1)} \\ [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots [\sigma \leftarrow \ell[1] \{\uparrow^\sigma\}^{(i-1)}] \cdots [\sigma \leftarrow \ell[i-1] \uparrow^\sigma] [\sigma \leftarrow \ell[i]] \end{array} \right\rangle \\
& \rightarrow^* \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle \{\uparrow^\sigma\}^{(i-1)} \\
& \quad [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots [\sigma \leftarrow \ell[1] \{\uparrow^\sigma\}^{(i-1)}] \cdots [\sigma \leftarrow \ell[i-1] \uparrow^\sigma] [\sigma \leftarrow \ell[i]] \\
& \rightarrow^* \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots \\
& \quad \{\uparrow^\sigma\}^{(i-1)} [\sigma \leftarrow \ell[1] \{\uparrow^\sigma\}^{(i-1)}] \cdots [\sigma \leftarrow \ell[i-1] \uparrow^\sigma] [\sigma \leftarrow \ell[i]] \\
& \rightarrow^* \langle M \uparrow^\sigma \mid \sigma \uparrow^* \rangle [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots [\sigma \leftarrow \ell[i]] \\
& \rightarrow^* \langle M [\gamma \leftarrow e_1 \uparrow^{\alpha\cdots\uparrow^\sigma}] [\alpha \leftarrow e_2 \uparrow^{\cdots\uparrow^\sigma}] \cdots \mid \ell[i] \rangle
\end{aligned}$$

While this command does have the intended effect, it also has some drawbacks. Obviously, it does not take constant time to perform the reductions. In fact, it shows that because the substitution elimination rules are only applied after all of the bindings have been created, the complexity of applying the (τe) rule is $O(n^2)$ where n is the number of weakenings(bindings). Also, the term M is bound deep inside the command. This is not a problem with normal control flow instructions, instructions that use the form $\text{do}@ (i)$, because the state is available where the term is formed. However, this does create a problem with instruction queues that end with $\text{do } r!$ because the state of the machine when the $@(i)$ expression was formed does not necessarily represent the state of the machine when the do is performed, and the bindings for γ and α , for example would already be made. We need to turn the command into a function that takes two values and applies one argument to the other. This is easily accomplished by simply

abstracting out ℓ and M . This leads to the function

$$\lambda x.\lambda y.\mu_{-}.\langle x \uparrow^y \mid \{\sigma\lambda.\{\}^{(i-1)}\sigma\lambda.\bar{\mu}_{-}.\langle y \uparrow^\sigma \mid \sigma \uparrow^y \rangle \{\uparrow^\sigma\}^{(i-1)} \uparrow^x \rangle \quad (\text{II.1})$$

As before, “ $\{\sigma\lambda.\{\}^{(i-1)}\}$ ” and “ $\{\uparrow^\sigma\}^{(i-1)}$ ” are replaced by $i - 1$ copies of “ $\sigma\lambda.\{\}$ ” and “ \uparrow^σ ”.)” The function takes a list that contains at least two terms. The first term must be a list of at least i contexts. The i^{th} context is supplied to the second term. The calling convention is $\langle \sigma \uparrow^* \mid \ell \cdot M \cdot e \rangle$ where \uparrow^* indicates that the expression should contain enough weakenings to destroy any substitutions in effect.

It is convenient to imagine a family of functions, $\hat{\sigma}$ that are created as needed, or alternatively, an unbounded number of them are automatically bound as dynamic functional continuations in the computation. They are dynamic in the sense that they are passed around by name rather than using the immediate substitution syntax. This eliminates the need for explicit weakenings, which would be an unacceptable computational burden, especially if every term were required to contain an infinite number of them. Henceforth, we will refer to the functional continuations $\hat{\sigma}_i$ such that $\langle \hat{\sigma}_i \mid \ell \cdot M \cdot e \rangle$ rewrites to $\langle M \mid (\ell[i])^\circ \rangle$. Furthermore, although we will use the form of the functions given above to obtain the proper type, we will consider that they in fact return in constant time. That is, the selection operator $\hat{\sigma}_i$ is an oracle for context list decomposition.

II.2.2 Extensions as Functional Continuations

In the last section, we developed the idea of dynamic functional continuations to model a family of functions. In that case, this was merely a notational convenience to allow us to bypass the bookkeeping on weakenings for all the bindings. However, it is also possible to use families of dynamic function continuations to extend the expressiveness of the calculus. For example, we might want to have a functional continuation $\hat{\sigma}$ such

$\langle \widehat{\sigma}_i \mid \ell \cdot M \cdot \epsilon \rangle$	\rightarrow	$\langle M \mid \ell[i] \rangle$	
$\langle \widehat{\sigma}_{i,j} \mid v \cdot \ell \cdot M \cdot \epsilon \rangle$	\rightarrow	$\langle M \mid \ell[i] \rangle$	$v = 0$
$\langle \widehat{\sigma}_{i,j} \mid v \cdot \ell \cdot M \cdot \epsilon \rangle$	\rightarrow	$\langle M \mid \ell[j] \rangle$	$v \neq 0$
$\langle \widehat{\tau}_A \mid v \cdot e \rangle$	\rightarrow	$\langle 1 \mid e \rangle$	$v : A$
$\langle \widehat{\tau}_A \mid v \cdot e \rangle$	\rightarrow	$\langle 0 \mid e \rangle$	$v : B \text{ where } A \neq B$

Figure II.6: Extension function reduction semantics

that $\langle + \mid v_1 \cdot v_2 \cdot e \rangle \rightarrow \langle i + j \mid e \rangle$. Although calculi exist that can model fixed precision arithmetic, it is much easier to add a family of arithmetic operators as functions that just work. Rather than produce the expressions for the functions and derive the reduction semantics and types, we simply give the reduction semantics and types and consider the functions as extensions to the calculus.

We use this idea to include the branch functions. For our work on the PPL machine, we need an oracle that will choose an appropriate address based on the status of r . We use the form $\widehat{\sigma}_{i,j}$ to be the function of three arguments that behaves like $\widehat{\sigma}_i$ if the first argument is zero or $\widehat{\sigma}_j$ otherwise. The formal semantics of this new oracle is given in Figure II.6 along with the normal selection operator, and the typing oracle. The dynamic typing functions are a family of functions $\widehat{\tau}_A$ parameterized by A that return a value to the second parameter based on the dynamic type of the first parameter.

II.2.3 Programs and Instructions

There are any number of approaches to code translation into the calculus. In representations of JVMIL, the `jsr/ret` pairs tend to be a stumbling block. In particular, when the `ret` instruction is executed, control returns to the calling code, but the local stack and environment do not return to the calling state. On the other hand, when a `return` instruction executes, the current stack and local variables are discarded in favor of the calling code's bindings. This multi-tiered bindings property prevents a straight-forward

representation of `jsr` as a jump with current continuation. Instead, we adopt a state passing style. Thus, we expect each instruction to bind to the current values of the PPL list registers, and then execute its internal commands. This suggests that each instruction queue have the form $\varphi\lambda.\gamma\lambda.\alpha\lambda.v$ where φ is the $\lambda\mu\bar{\mu}\uparrow$ context equivalent of Φ and v does the work of the instruction. This creates a correspondence between instruction queues and contexts. Also, since \mathcal{P} corresponds to a list of instruction queues, it can in turn be represented as a list of contexts s , which is a term in this calculus. Since computational content in $\lambda\mu\bar{\mu}\uparrow$ generally corresponds to commands, for technical reasons we require each encoded instruction queue have the form $\varphi\lambda.\gamma\lambda.\alpha\lambda.\bar{\mu}s.c$. Thus, the `do` instruction will be required to feed φ , γ , α and s to the new instruction queue. Because we expect to package and unpackage the state object frequently, we will adopt the following abbreviations:

$$\begin{aligned} \mathcal{M} &\triangleq \varphi \uparrow^{-\varphi} \cdot \gamma \uparrow^{-\gamma} \cdot \alpha \uparrow^{-\alpha} \cdot s \uparrow^{-s} \\ \overline{\mathcal{M}}(c) &\triangleq (\varphi\lambda.\gamma\lambda.\alpha\lambda.\bar{\mu}s. \langle \lambda r.\mu\delta.c \mid (0 \cdot \epsilon) \uparrow^{\{\varphi,\gamma,\alpha,s\}} \rangle) \\ \mathcal{M}^r &\triangleq r \uparrow^{-r} \cdot (\mathcal{M}) \cdot \epsilon \uparrow^* \\ \overline{\mathcal{M}}^r(c) &\triangleq (\lambda r.\lambda s.\mu\bar{\mu}. \langle s \uparrow^r \mid (\varphi\lambda.\gamma\lambda.\alpha\lambda.\bar{\mu}s. \langle \mu\delta.c \mid \epsilon \uparrow^{-\delta} \rangle) \uparrow^s \rangle) \end{aligned}$$

The first form, \mathcal{M} is the state constructor, whereas the second form, $\overline{\mathcal{M}}()$ is an apparatus for deconstruction. In addition to assigning the variables from the state, $\overline{\mathcal{M}}()$ also assigns the default values to r and δ . Rather than constantly passing 0 and ϵ in the state, we just leave them out and assume those values for r and δ during a transition from one instruction to the next. The expression $\langle \mathcal{M} \mid \overline{\mathcal{M}}(c) \rangle$ packages the state into a term, then unpacks it and executes the command c . While this exact form is unlikely in our translations, the pattern is an important one. The form $\overline{\mathcal{M}}^r()$ is necessary to pass values

back from `restore`. This only assigns the default value to δ , but takes the value for r as a parameter. The form $\langle \overline{\mathcal{M}^r}(c) \mid \mathcal{M}^r \rangle$ passes everything but δ to c .

Example II.2.1 *The command $\langle \widehat{\sigma}_i \uparrow^* \mid (s \uparrow^{-s}) \cdot \mathcal{M} \cdot (\epsilon \uparrow^*) \rangle$ correctly selects the i^{th} encoded instruction queue, and continues the computation with the given state.*

We let $(\mathcal{P})^\circ$ be a list of contexts, the i^{th} entry of which is $\overline{\mathcal{M}}(c)$. For this to properly reduce, the command must be issued at a point in the computation with bindings for all the PPL structures. The reduction sequence with the proper weakenings is as follows:

$$\begin{aligned}
& \langle \widehat{\sigma}_i \uparrow^* \mid s \uparrow^{-s} \cdot \varphi \uparrow^{-\varphi} \cdot \gamma \uparrow^{-\gamma} \cdot \alpha \uparrow^{-\alpha} \cdot s \uparrow^{-s} \cdot \epsilon \uparrow^* \rangle \\
& \quad [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^s \uparrow^\delta \uparrow^r] [\gamma \leftarrow g \uparrow^\alpha \uparrow^s \uparrow^\delta \uparrow^r] [\alpha \leftarrow a \uparrow^s \uparrow^\delta \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow d \uparrow^r] [r \leftarrow 27] \\
& \rightarrow^* \langle \widehat{\sigma}_i \mid (\mathcal{P})^\circ \cdot f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon \rangle \\
& \rightarrow^* \langle f \cdot g \cdot a \cdot (\mathcal{P})^\circ \mid \varphi \lambda. \gamma \lambda. \alpha \lambda. \bar{\mu} s. \langle \lambda r. \mu \delta. c \mid (0 \cdot \epsilon) \uparrow^* \rangle \rangle \\
& \rightarrow^* \langle g \cdot a \cdot (\mathcal{P})^\circ \mid \gamma \lambda. (\alpha \lambda. \bar{\mu} s. \langle \lambda r. \mu \delta. c \mid (0 \cdot \epsilon) \uparrow^* \rangle) [\varphi \leftarrow f \uparrow^\gamma] \rangle \\
& \rightarrow^* \langle a \cdot (\mathcal{P})^\circ \mid \alpha \lambda. (\bar{\mu} s. \langle \lambda r. \mu \delta. c \mid (0 \cdot \epsilon) \uparrow^* \rangle) [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha] [\gamma \leftarrow g \uparrow^\alpha] \rangle \\
& \rightarrow^* \langle (\dots \cdot (\mathcal{P})^\circ \mid \bar{\mu} s. (\langle \lambda r. \mu \delta. c \mid (0 \cdot \epsilon) \uparrow^* \rangle) [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^s] [\gamma \leftarrow g \uparrow^\alpha \uparrow^s] [\alpha \leftarrow a \uparrow^s]) \rangle \\
& \rightarrow^* \langle \lambda r. \mu \delta. c \mid (0 \cdot \epsilon) \uparrow^* \rangle [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^s] [\gamma \leftarrow g \uparrow^\alpha \uparrow^s] [\alpha \leftarrow a \uparrow^s] [s \leftarrow (\mathcal{P})^\circ] \\
& \rightarrow^* c [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] [\gamma \leftarrow g \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] [\alpha \leftarrow a \uparrow^s \uparrow^r \uparrow^\delta] [s \leftarrow (\mathcal{P})^\circ \uparrow^r \uparrow^\delta] [r \leftarrow 0 \uparrow^\delta] [\delta \leftarrow \epsilon]
\end{aligned}$$

Using the expanded, explicit form of the selection operator, the second step can be seen as

$$\begin{aligned}
& \rightarrow^* \langle \widehat{\sigma}_i \mid (\mathcal{P})^\circ \cdot f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon \rangle \\
& = \left\langle \lambda x. \lambda y. \mu \bar{\mu} \left\langle x \uparrow^y \mid \{ \sigma \lambda. \{ \}^{(i-1)} \sigma \lambda. \bar{\mu} \bar{\mu} \langle y \uparrow^\sigma \mid \sigma \uparrow^y \rangle \{ \uparrow^\sigma \}^{(i-1)} \uparrow^x \} \mid (\mathcal{P})^\circ \cdot f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon \right\rangle \right. \\
& \rightarrow^* \left\langle \begin{array}{l} (\mathcal{P})^\circ \mid \bar{\mu} x. \langle \lambda y. \mu \bar{\mu} \langle x \uparrow^y \mid \{ \sigma \lambda. \{ \}^{(i-1)} \sigma \lambda. \bar{\mu} \bar{\mu} \langle y \uparrow^\sigma \rangle \\ \mid \sigma \uparrow^y \rangle \{ \uparrow^\sigma \}^{(i-1)} \uparrow^x \} \mid (f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon) \uparrow^x \end{array} \right\rangle \\
& \rightarrow^* \left\langle \begin{array}{l} \lambda y. (\mu \bar{\mu} \langle x \uparrow^y \mid \{ \sigma \lambda. \{ \}^{(i-1)} \sigma \lambda. \bar{\mu} \bar{\mu} \langle y \uparrow^\sigma \rangle \\ \mid \sigma \uparrow^y \rangle \{ \uparrow^\sigma \}^{(i-1)} \uparrow^x \} [x \leftarrow (\mathcal{P})^\circ \uparrow^y]) \mid f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon \end{array} \right\rangle
\end{aligned}$$

$$\begin{aligned}
& \rightarrow^* \left\langle \begin{array}{l} f \cdot g \cdot a \cdot (\mathcal{P})^\circ \\ | \bar{\mu}y. \langle \mu_. \langle x \uparrow^y \mid \{\sigma\lambda.(\cdot)^{(i-1)} \sigma\lambda. \bar{\mu}_. \langle y \uparrow^\sigma \mid \sigma \uparrow^y \rangle \{\uparrow^\sigma\} \}^{(i-1)} \uparrow^x \rangle [x \leftarrow (\mathcal{P})^\circ \uparrow^y] \mid \epsilon \uparrow^y \rangle \end{array} \right\rangle \\
& \rightarrow^* \left\langle \begin{array}{l} \mu_. \langle x \uparrow^y \mid \{\sigma\lambda.(\cdot)^{(i-1)} \sigma\lambda. \bar{\mu}_. \langle y \uparrow^\sigma \\ | \sigma \uparrow^y \rangle \{\uparrow^\sigma\} \}^{(i-1)} \uparrow^x \rangle [x \leftarrow (\mathcal{P})^\circ \uparrow^y] [y \leftarrow f \cdot g \cdot a \cdot (\mathcal{P})^\circ] \mid \epsilon \end{array} \right\rangle \\
& \rightarrow^* \langle (\mathcal{P})^\circ \mid \{\sigma\lambda.(\cdot)^{(i-1)} \sigma\lambda. \bar{\mu}_. \langle y \uparrow^\sigma \mid \sigma \uparrow^y \rangle \{\uparrow^\sigma\} \}^{(i-1)} [y \leftarrow f \cdot g \cdot a \cdot (\mathcal{P})^\circ] \rangle \\
& \rightarrow^* \langle y \uparrow^\sigma \mid \sigma \uparrow^y \rangle [y \leftarrow f \cdot g \cdot a \cdot (\mathcal{P})^\circ] [\sigma \leftarrow (\mathcal{P}[i])^\circ] \\
& \rightarrow^* \langle f \cdot g \cdot a \cdot (\mathcal{P})^\circ \mid (\mathcal{P}[i])^\circ \rangle \\
& = \langle f \cdot g \cdot a \cdot (\mathcal{P})^\circ \mid \varphi\lambda.\gamma\lambda.\alpha\lambda.\bar{\mu}s. \langle \lambda r. \mu\delta.c \mid (0 \cdot \epsilon) \uparrow^* \rangle \rangle
\end{aligned}$$

In the example, we performed the same action as the PPL do command:

$$\left(\text{do } @ (i)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}, \Phi \rangle \right) \xrightarrow{\text{PPL}} \left(\mathcal{P}[i], 0, s_1, \odot, s_3, \langle \mathcal{P}, \Phi \rangle \right)$$

Where the selection operator $\hat{\sigma}_i$ represents the expression $@(i)$, and also computes $\mathcal{P}[i]$ from a given \mathcal{P} , represented here as s . γ and α have retained their initial values, as have φ , which is the analog of Φ , and s , which is the analog of \mathcal{P} . The current calculation has become the i^{th} element of s , which is the equivalent of setting $I = \mathcal{P}[i]$, and δ and r have been set to the empty list and zero, respectively. If we view the code store as a list of continuations, then the behavior of the system is reminiscent of CPS, where each instruction performs its transformation on the state and then passes it to the next instruction. By providing us with the ability to create a context as a list of terms followed by a context and vice versa, $\lambda\mu\bar{\mu}\uparrow$ supports the inclusion of arbitrary information into a term.

Since the action of each instruction can be seen as performing some transformation on the state term, we see that the instruction form $\varphi\lambda.\gamma\lambda.\alpha\lambda.\bar{\mu}s.c$ is necessary to access the term's fields, bearing in mind, of course, that a PPL instruction queue may represent

more than one JVMIL instruction if the control flow transformation from section I.4 is used. It is only necessary to unwrap the state once at the beginning of the instruction queue, and repackage it once at the end of the queue. However, since I is consumed in the process of computation, when we represent the state, we will want to show I as a simple command. Thus, if we have a function $(I)^*$ that translates a PPL instruction queue into a $\lambda\mu\tilde{m}\uparrow$ expression, we extend it to an array of instruction queues by prepending each queue in the array with the state accessor apparatus. Since the resulting expression is a context, we can form a term by using the \cdot (cons) operator on the individual contexts and terminating the whole list with ε . More formally, the extension of the instruction queue translation function to instruction queue arrays is given as:

$$\begin{aligned} (\mathcal{P})^\circ &= (\mathcal{P})^\circ \cdot \varepsilon \\ (\mathcal{P}[1..i])^\circ &= (\mathcal{P}[1..i-1])^\circ \cdot \overline{\mathcal{M}}((\mathcal{P}[i])^*) \end{aligned}$$

Here, we have delegated clearing r and δ to the instruction queue initialization rather than to the finalization of the previous instruction queue. This is simply a matter of convenience, the validity of which is guaranteed by Lemma I.3.4.

Although the translation is just the identity function for primitive values, there are a few values that must be given assignment:

$$\begin{aligned} (\odot)^\circ &= \epsilon \\ (\boxtimes)^\circ &= \varepsilon \\ (@(n))^\circ &= \widehat{\sigma}_n \end{aligned}$$

bearing in mind that $\widehat{\sigma}_n$ can be formally defined as the inlined function macro shown in section II.2.1, but we choose to think of it as a dynamically bound functional continuation. Whichever way it is defined, we must be certain that we don't pick up a stray

redex that causes it to be evaluated rather than stored in r . In particular, we want the selection function, not the particular code that it refers to, because storing the code at this point will cause the JVM structures to be bound to their current values; whereas, “loading” the instruction later allows us to use the JVM structures as they exist at that time.

Example II.2.2 *The command*

$$\langle \hat{\sigma}_i \uparrow^* \mid (\bar{\mu}r.c) \uparrow^r \rangle$$

correctly binds $\hat{\sigma}_i$ to r in c . We let

$$c_{s_i} = \langle x \mid (\sigma\lambda.)^{(i)} \bar{\mu}_-. (y \uparrow^\sigma \mid \sigma) (\uparrow^\sigma)^{(i-1)} \rangle$$

in the reduction sequence:

$$\begin{aligned} & \langle \hat{\sigma}_i \uparrow^* \mid (\bar{\mu}r.c) \uparrow^r \rangle \\ & \quad [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\gamma \leftarrow g \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\alpha \leftarrow a \uparrow^\sigma \uparrow^\delta \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow d \uparrow^r] [r \leftarrow 27] \\ \rightarrow^* & \langle \hat{\sigma}_i \mid \bar{\mu}r.(c [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\gamma \leftarrow g \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\alpha \leftarrow a \uparrow^\sigma \uparrow^\delta \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow d \uparrow^r]) \rangle \\ = & \left\langle \begin{array}{l} (\lambda x.\lambda y.\bar{\mu}_-.c_{s_i}) \mid \bar{\mu}r.(c [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] \\ [\gamma \leftarrow g \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\alpha \leftarrow a \uparrow^\sigma \uparrow^\delta \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow d \uparrow^r]) \end{array} \right\rangle \end{aligned}$$

This does not create a call-by-name/call-by-value conflict because the rule (β_-) cannot be applied to this context. Thus, the redex goes as planned:

$$\rightarrow^* c [\varphi \leftarrow f \uparrow^\gamma \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\gamma \leftarrow g \uparrow^\alpha \uparrow^\sigma \uparrow^\delta \uparrow^r] [\alpha \leftarrow a \uparrow^\sigma \uparrow^\delta \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow d \uparrow^r] [r \leftarrow \hat{\sigma}_i \uparrow^*]$$

Example II.2.3 *The command*

$$\langle r \uparrow^{-r} \mid (s \uparrow^{-s}) \cdot \mathcal{M} \cdot (\epsilon \uparrow^*) \rangle$$

correctly transforms a state with r bound to $\hat{\sigma}_i$ into a command of the form in Example II.2.1.

$$\begin{aligned} & \langle r \uparrow^{-r} \mid s \uparrow^{-s} \cdot \varphi \uparrow^{-\varphi} \cdot \gamma \uparrow^{-\gamma} \cdot \alpha \uparrow^{-\alpha} \cdot s \uparrow^{-s} \cdot \epsilon \uparrow^* \rangle \\ & \quad [\varphi \leftarrow f \uparrow^r \uparrow^{\alpha} \uparrow^s \uparrow^{\delta} \uparrow^r] [\gamma \leftarrow g \uparrow^{\alpha} \uparrow^s \uparrow^{\delta} \uparrow^r] [\alpha \leftarrow a \uparrow^s \uparrow^{\delta} \uparrow^r] [s \leftarrow (\mathcal{P})^\circ \uparrow^{\delta} \uparrow^r] [\delta \leftarrow d \uparrow^r] [r \leftarrow \hat{\sigma}_i] \\ \rightarrow & \langle \hat{\sigma}_i \mid (\mathcal{P})^\circ \cdot f \cdot g \cdot a \cdot (\mathcal{P})^\circ \cdot \epsilon \rangle \end{aligned}$$

The rest of the reduction proceeds as in Example II.2.1.

Theoretically, Φ holds a complete state, but we know that every state saved in Φ has $r = 0$ and $\delta = \odot$. Therefore, we can leave out the encoding of those values and just substitute the constants whenever we restore the state. On the other hand, Φ does contain an Φ , so we have the following inductive definition of the translation $(\Phi)^\circ$ of Φ into a $\lambda\mu\bar{\mu}\uparrow$ context:

$$\begin{aligned} (\odot)^\circ &= \epsilon \uparrow^* \\ \left(\left(I_0, r_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \right)^\circ &= ((\Phi_0)^\circ \cdot (\gamma_0)^\circ \cdot (\alpha_0)^\circ \cdot (\mathcal{P}_0)^\circ) \cdot \overline{\mathcal{M}}^r ((I_0)^*) \cdot \epsilon \end{aligned}$$

and the following definition for the translation of a PPL state into a $\lambda\mu\bar{\mu}\uparrow$ command:

$$\left(\left(I_0, r_0, \alpha_0, \gamma_0, \delta_0, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \right)^\circ = (I_0)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^r \uparrow^{\alpha} \uparrow^s \uparrow^{\delta} \uparrow^r] \\ [\gamma \leftarrow (\gamma_0)^\circ \uparrow^{\alpha} \uparrow^s \uparrow^{\delta} \uparrow^r] \\ [\alpha \leftarrow (\alpha_0)^\circ \uparrow^s \uparrow^{\delta} \uparrow^r] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^{\delta}] \\ [r \leftarrow (r_0)^\circ \uparrow^{\delta}] \\ [\delta \leftarrow (\delta_0)^\circ] \end{pmatrix}$$

The translations for the PPL instructions are given in Figure II.7. The PPL macros `transfer` and `dig` are expanded before translation, so they are not listed in the translation.

$(\text{const } v; c!)^*$	$= \langle (v)^\circ \uparrow^* \mid (\tilde{\mu}r.(c!)^*) \uparrow^r \rangle$
$(\text{isa}_A; c!)^*$	$= \langle \widehat{\tau}_A \uparrow^* \mid r \uparrow^{-r} \cdot (\tilde{\mu}r.(c!)^*) \uparrow^r \rangle$
$(\text{mustbe } A; c!)^*$	$= \langle \mu_{-}.(c!)^* \mid \tilde{\mu}_{-} \langle r \uparrow^{-r} \mid \llbracket \! \! \! _A \uparrow^* \rangle \rangle$
$(\text{push}^k; c!)^*$	$= \langle (\mu k.(c!)^*) \uparrow^k \mid (r \uparrow^{-r}) \cdot (k \uparrow^{-k}) \rangle$
$(\text{pop}^k; c!)^*$	$= \langle (\lambda r. \mu k.(c!)^*) \uparrow^r \uparrow^k \mid k \uparrow^{-k} \rangle$
$(\text{dump}^\circ; c!)^*$	$= \langle (\mu \delta.(c!)^*) \uparrow^\delta \mid \epsilon \uparrow^* \rangle$
$(\text{dump}^k; c!)^*$	$= \langle (\mu \delta.(c!)^*) \uparrow^\delta \mid k \uparrow^{-k} \rangle$
$(\text{retrieve}^k; c!)^*$	$= \langle (\mu k.(c!)^*) \uparrow^k \mid \delta \uparrow^{-\delta} \rangle$
$(\text{call } (\mathcal{P}); c!)^*$	$= \left\langle \begin{array}{l} \mathcal{M} \cdot (\overline{\mathcal{M}}^r ((c!)^*) \uparrow^*) \cdot \epsilon \uparrow^* \cdot \delta \uparrow^{-\delta} \cdot \epsilon \uparrow^* \cdot (\mathcal{P})^\circ \uparrow^* \\ \mid \overline{\mathcal{M}} ((\text{do } @ (1)!)^*) \uparrow^\varphi \uparrow^r \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta \end{array} \right\rangle$
$(\text{restore}!)^*$	$= \langle (\lambda s. \lambda t. \mu_{-} \langle t \uparrow^* \mid r \uparrow^{-r} \uparrow^t \cdot s \uparrow^{-s} \uparrow^t \cdot \epsilon \uparrow^* \uparrow^t \rangle) \uparrow^s \mid \varphi \uparrow^{-\varphi} \rangle$
$(\text{do } r!)^*$	$= \langle r \uparrow^{-r} \mid s \uparrow^{-s} \cdot \mathcal{M} \cdot \epsilon \uparrow^* \rangle$
$(\text{do } @ (n)!)^*$	$= \langle \widehat{\sigma}_n \uparrow^* \mid s \uparrow^{-s} \cdot \mathcal{M} \cdot \epsilon \uparrow^* \rangle$
$(\text{do } @ (n)? @ (m)!)^*$	$= \langle \widehat{\sigma}_{n,m} \uparrow^* \mid r \uparrow^{-r} \cdot s \uparrow^{-s} \cdot \mathcal{M} \cdot \epsilon \uparrow^* \rangle$

Figure II.7: Translation from PPL to $\lambda\mu\tilde{\mu}\uparrow$ -calculus

const This encoding simply binds the new value to r

isa This encoding uses the typing function extension to place the proper value in r

mustbe This encoding uses two anonymous variables to construct a term that has no computational side-effects, but will not be well-typed in a call-by-value system unless r contains a value with the correct type. Here, we have to be careful about specifying a call-by-value strategy because giving priority to μ will cause the command to reduce to c , the intended computation, rather than $\langle r \mid \llbracket \! \! \! _A \rangle$, a finished computation. Of course, under a call-by-name strategy, we could have used the command

$$\langle \mu_{-} \langle r \uparrow^{-r} \mid \llbracket \! \! \! _A \rangle \mid \tilde{\mu}_{-}.(c!)^* \rangle$$

which discards the term binding. This makes use of the fact that the typing system must type the anonymous variable, even though it does not contribute to the computation.

push / pop This encoding uses the term list constructor and destructor to bind/unbind r to/from the front of the given context.

dump / retrieve This encoding simply binds the context from one context variable to another context variable.

call This encoding binds the state and remaining instruction queue to φ , following the semantics of $()^*$, then associates the new code array with s and executes the first instruction. We store the code using $\overline{\mathcal{M}}^r()$ in anticipation of receiving a value for r from the restore command.

- `restore` This encoding performs the inverse of the `call` command, dropping the current state and recovering the previous state from φ . This command uses the fresh temporary variable t to hold the restored command, but the binding for t does not persist into the computation. The outer command decomposes φ into $s = \mathcal{M}$ and $t = \overline{\mathcal{M}^r}()$. The inner command constructs $\mathcal{M}^r = r \cdot \mathcal{M} \cdot \epsilon$ and supplies it to $\overline{\mathcal{M}^r}()$.
- `do` This encoding uses the selection operators to execute the given instruction queue. The specific forms come from the examples.

II.3 Proving the Correctness of the Translation

II.3.1 Weakened Normal Forms

Like the function $()^\triangleright$ from JVM states to PPL states, the function $()^\circ$ from PPL states to $\lambda\mu\tilde{\mu}\dagger$ expressions is not invertible. That is, there are many $\lambda\mu\tilde{\mu}\dagger$ expressions that cannot be generated from valid PPL states via the $()^\circ$ function. However, unlike $()^\triangleright$, many $\lambda\mu\tilde{\mu}\dagger$ expressions are equivalent to the translation in the sense that they reduce to common expressions in the same number of steps. Since a computation may reduce to any of these equivalent expressions, we need a framework for recognizing two expressions that differ only in unimportant ways.

In particular, the order of substitutions on the top of a redex will vary based on the command sequence performed, but our translation function produces a strict canonical order that does not take this into account. Thus, we must define an equivalence class of $\lambda\mu\tilde{\mu}\dagger$ terms that correspond to the same PPL state modulo ordering of the outer substitutions. The idea is to allow the outer substitutions to occur in any order, but they still have to carry the proper weakenings, which is a function of their order.

Definition II.3.1 *Weakened Normal Form (WNF) is defined inductively as follows:*

- $[x \leftarrow v]$ and $[k \leftarrow e]$ are in Weakened Normal Form (WNF) provided v and e contain no outer weakenings.
- If u is in weakened normal form, then for all $x \notin \text{dom}(u)$ and $k \notin \text{dom}(u)$
 - $(u \ [“J” \ “J \uparrow^x ”]) \ [x \leftarrow v]$ and
 - $(u \ [“J” \ “J \uparrow^k ”]) \ [k \leftarrow e]$

are also in weakened normal form.

Here, the $[“t” \ “s”]$ operator means textual substitution of “s” for “t”, and $\text{dom}(u)$ represents the set of variables for which u contains a substitution.

This definition says that

$$[\alpha \leftarrow A \uparrow^\gamma] [\gamma \leftarrow B] \text{ and } [\alpha \leftarrow A \uparrow^\gamma \uparrow^\delta] [\gamma \leftarrow B \uparrow^\delta] [\delta \leftarrow C]$$

are WNF, but

$$[\alpha \leftarrow A] [\gamma \leftarrow B], \quad [\alpha \leftarrow \gamma] [\gamma \leftarrow B], \quad [\alpha \leftarrow A_1 \uparrow^\alpha] [\alpha \leftarrow A_2]$$

$$\text{and } [\alpha \leftarrow A \uparrow^\delta \uparrow^\gamma] [\gamma \leftarrow C \uparrow^\delta] [\delta \leftarrow D]$$

are not. Obviously, WNF is a purely syntactic notion that is not intended to capture anything as powerful as semantic well-formedness. In the remaining part of this section, we will describe the equivalence classes of WNF formulae that have the same effect on commands, and then extend the relation to form equivalence classes of commands.

The following two lemmas are needed to show that the equivalence classes of WNF substitutions are well-defined.

Lemma II.3.1 *If u is a list of substitutions in WNF and k is a continuation variable with a substitution $[k \leftarrow e \ U]$ in u then $k \uparrow^{dom(u) \setminus k} u \rightarrow e$.*

Proof. Proof is by structural induction on u , maintaining the invariant that the expression has the form $k \uparrow^{dom(u) \setminus k} u$ or $e \uparrow^{dom(u)} u$.

- **Case $u = [x \leftarrow v] \ u'$:** By the invariant, the context is either
 - $k \uparrow^{dom(u) \setminus k} u$: Since $x \neq k$, the weakening contains x . Thus, by rule (τv) , this reduces to $k \uparrow^{dom(u) \setminus k \setminus x} u' = k \uparrow^{dom(u') \setminus k}$
 - $e \uparrow^{dom(u)} u$: Since $x \in dom(u')$, by (τv) , $e \uparrow^{dom(u)} [x \leftarrow v] \ u' \rightarrow e \uparrow^{dom(u) \setminus x} u' = e \uparrow^{dom(u')} \ u'$
- **Case $u = [k' \leftarrow e] \ u'$:** This case is similar to the previous one.
- **Case $u = [k \leftarrow e \ U] \ u'$:** If u contains a substitution for k , then k must still be on the left of the context (since u can contain at most one substitution for k). By the definition of WNF U must contain a weakening for each variable in $dom(u')$. That is, $U = \uparrow^{dom(u) \setminus k}$. By the invariant, we have the context $k \uparrow^{dom(u) \setminus k} [k \leftarrow e \ \uparrow^{dom(u) \setminus k}] \ u'$ which factors to $k [k \leftarrow e] \ \uparrow^{dom(u) \setminus k} \ u' = k [k \leftarrow e] \ \uparrow^{dom(u')} \ u' \rightarrow e \uparrow^{dom(u')} \ u'$
- **Case $u = \emptyset$:** Since the only way to have eliminated $[k \leftarrow e \ U]$ from u would have been using rule $(k\tau)$, which would have placed e on the left, we must be left with just e , as required.

■

Lemma II.3.2 *If u is a list of substitutions in WNF and x is a regular variable with a substitution $[x \leftarrow v \ U]$ in u then $x \uparrow^{dom(u) \setminus x} u \rightarrow v$.*

Proof. The same as for Lemma II.3.1, mutatis mutandis ■

We can now define the equivalence relation \sim_{\uparrow} over substitutions in WNF as $u \sim_{\uparrow} u'$ provided the following hold:

- both u and u' are WNF
- $dom(u) = dom(u')$
- for any continuation variable $k \in dom(u)$, $k \uparrow^{dom(u) \setminus k} u \rightarrow^* v$ if and only if $k \uparrow^{dom(u) \setminus k} u' \rightarrow^* v$
- for any regular variable $x \in dom(u)$, $x \uparrow^{dom(u) \setminus x} u \rightarrow^* v$ if and only if $x \uparrow^{dom(u) \setminus x} u' \rightarrow^* v$

For example, $[\alpha \leftarrow a \uparrow^{\gamma}] [\gamma \leftarrow b] \sim_{\uparrow} [\gamma \leftarrow b \uparrow^{\alpha}] [\alpha \leftarrow a]$ since

$$\alpha \uparrow^{\gamma} [\alpha \leftarrow a \uparrow^{\gamma}] [\gamma \leftarrow b] = a = \alpha \uparrow^{\gamma} [\gamma \leftarrow b \uparrow^{\alpha}] [\alpha \leftarrow a]$$

and

$$\gamma \uparrow^{\alpha} [\alpha \leftarrow a \uparrow^{\gamma}] [\gamma \leftarrow b] = b = \gamma \uparrow^{\alpha} [\gamma \leftarrow b \uparrow^{\alpha}] [\alpha \leftarrow a]$$

Next, we extend \sim_{\uparrow} to an equivalence class over commands by letting $c \sim_{\uparrow} c'$ if and only if $c = c'$ and $u \sim_{\uparrow} u'$. That is, two commands are equivalent if the only difference is the order of outer substitutions, which must be WNF. Obviously, \sim_{\uparrow} is an equivalence relation that defines the equivalence classes

$$[c]_{\sim_{\uparrow}} = \{c' \mid c' \sim_{\uparrow} c\}$$

Intuitively, $[c]_{\sim_{\uparrow}}$ is the set of commands that are the same modulo the order of substitutions. Lemmas II.3.1 and II.3.2 guarantee that the order of substitutions does not matter. To show that reduction preserves equivalence, we need the following lemma.

Lemma II.3.3 *If $u_1 \sim_{\uparrow} u_2$, then for every $x \in \text{dom}(u_1)$ and $k \in \text{dom}(u_1)$, $c \uparrow^x u_1 \sim_{\uparrow} c \uparrow^x u_2$ and $c \uparrow^k u_1 \sim_{\uparrow} c \uparrow^k u_2$*

Proof. We show the result for $c \uparrow^x u_1 \sim_{\uparrow} c \uparrow^x u_2$, with the conjunct involving context variables having a similar proof. We need to show that $x' \uparrow^x u_1 = x' \uparrow^x u_2$. If $x \in u_1$, then u_1 has the form $u'_1 [x \leftarrow v] u''_1$ where either u'_1 or u''_1 may be empty, but every substitute value and context in $\text{codom}(u'_1)$ contains a weakening for x , and no substitute value or context in $\text{codom}(u''_1)$ contains a weakening for x . Thus, \uparrow^x factors out of u'_1 and eliminates $[x \leftarrow v]$. Therefore, $\uparrow^x u_1$ reduces to a set of substitutions u_3 that are WNF and contain all the substitutions in u_1 except x . The same argument holds for $\uparrow^x u_2 \rightarrow^* u_4$. Thus, u_3 and u_4 perform the same action on x' , so $x' u_3 = x' u_4$ and $x' \uparrow^x u_1 = x' \uparrow^x u_2$. ■

We are now ready for the main result that reduction preserves WNF equivalence.

Theorem II.3.4 *For $cu_1 \sim_{\uparrow} cu_2$, if $cu_1 \rightarrow^* c'u'_1$ then there exists a command $c'u'_2$ such that $cu_2 \rightarrow^* c'u'_2$ and $c'u'_1 \sim_{\uparrow} c'u'_2$*

Proof. If $cu_1 \sim_{\uparrow} cu_2$, then $u_1 \sim_{\uparrow} u_2$. The proof proceeds by structural induction on $\langle v \mid e \rangle$. The following case is illustrative:

- **Case $v = x \uparrow^V, e = (\bar{\mu}x'.c_0) \uparrow^{x'}$:** By Lemma II.3.2, $x \rightarrow v_0$ for some v_0 under both sets of substitutions, or $x \uparrow^V u_1$ does not have sufficient weakenings to reduce further. If $x \uparrow^V u_1$ is not a proper redex, then the existence of c' is contradicted. Thus, $\langle v \mid e \rangle u_1 \rightarrow^* \langle v_0 \mid (\bar{\mu}x'.c_0) \uparrow^{x'} u_1 \rangle$ and $\langle v \mid e \rangle u_2 \rightarrow^* \langle v_0 \mid (\bar{\mu}x'.c_0) \uparrow^{x'} u_2 \rangle$. If $x' \notin \text{dom}(u_1)$, then $(\bar{\mu}x'.c_0) \uparrow^{x'} u_1$ is not a redex, and again the existence of c' is

contradicted. By an argument similar to Lemma II.3.3, $\uparrow^{x'} u_1 \rightarrow^* u'_1$ and $\uparrow^{x'} u_2 \rightarrow^* u'_2$ where u'_1 is just u_1 without the substitution for x' , and similarly for u'_2 . Thus, $\langle v_0 \mid (\tilde{\mu}x'.c_0) \uparrow^{x'} u_1 \rangle \rightarrow^* c_0 u'_1 [x' \leftarrow v_0]$ and $\langle v_0 \mid (\tilde{\mu}x'.c_0) \uparrow^{x'} u_2 \rangle \rightarrow^* c_0 u'_2 [x' \leftarrow v_0]$. Since $c_0 = c_0$, it remains only to be shown that $u'_1 [x' \leftarrow v_0] \sim_{\uparrow} u'_2 [x' \leftarrow v_0]$. Clearly, $x' \uparrow^{dom(u'_1) \setminus x'} u'_1 [x' \leftarrow v_0] = x' \uparrow^{dom(u'_2) \setminus x'} u'_2 [x' \leftarrow v_0] = v_0$, and for all other variables, the substitutions are unchanged. thus, $c' = c_0$.

The other cases are similar. ■

II.3.2 Confluence of Translated Instructions

In this section, we would like to show that translation followed by reduction is the same as reduction followed by translation. Unfortunately, for technical reasons, that is not the case. However, the previous section leads us to believe that the process is nonetheless confluent. To show this, we must appeal to the $\xrightarrow{\text{PPLI}}$ notation we adopted to prove Theorem I.3.5 That is, we would like to show the validity of the diagram

$$\begin{array}{ccccccc}
 S & \xrightarrow{\text{PPLI}} & S' & \xrightarrow{\text{PPLI}} & S'' & \xrightarrow{\text{PPLI}} & S''' \xrightarrow{\text{PPLI}} \dots \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 (S)^\circ & \longrightarrow & (S')^\circ & \longrightarrow & (S'')^\circ & \longrightarrow & (S''')^\circ \longrightarrow \dots
 \end{array}$$

where the \mapsto arrows represent translation from PPL states to $\lambda\mu\tilde{\mu}\uparrow$ commands. The next theorem almost gives us the diagram.

Theorem II.3.5 *For any PPL state S , if $S \xrightarrow{\text{PPLI}} S'$, then there exists a command c_0 such that $(S)^\circ \rightarrow^* c_0$ and $(S')^\circ \sim_{\uparrow} c_0$.*

For the use of \sim_{\uparrow} in the theorem to make sense, we need the next lemma.

Lemma II.3.6 *For any valid PPL state S , $(S)^\circ$ is WNF.*

Proof. By inspection on the output of $(S)^\circ$. ■

Since we must produce c_0 WNF, we can now prove II.3.5.

Proof. The proof proceeds by structural induction on the PPL instruction queues. In each case, we demonstrate the appropriate choice of c_0 .

- **Case const $v; c!$:**

$$\begin{aligned}
S &= \left(\text{const } v; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(cl, v, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S' \\
(S')^\circ &= (cl)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v)^\circ \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \sim_{\uparrow} (cl)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ \uparrow r] \\ [r \leftarrow (v)^\circ] \end{pmatrix} = c_0 \\
(S)^\circ &= \langle (v)^\circ \uparrow^* \mid (\bar{\mu}r. (cl)^*) \uparrow^r \rangle \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \quad [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \quad [r \leftarrow (v_0)^\circ \uparrow \delta] \quad [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \\
&\rightarrow^* \langle (v)^\circ \mid \bar{\mu}r. \left((cl)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \quad [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \quad [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \quad [\delta \leftarrow (s_3)^\circ \uparrow r] \end{pmatrix} \right) \rangle \\
&\rightarrow^* (cl)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \quad [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \quad [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \quad [\delta \leftarrow (s_3)^\circ \uparrow r] \quad [r \leftarrow (v)^\circ] \end{pmatrix} \\
&= c_0
\end{aligned}$$

- Case isa_τ :

- Case $v_0 : \tau$:

$$\begin{aligned}
S &= \left(isa_\tau; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(c!, 0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S' \\
(S')^\circ &= (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow 0 \uparrow \delta] \\ [\delta \leftarrow s_3] \end{pmatrix} \sim_{\uparrow} (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [\delta \leftarrow s_3 \uparrow r] \\ [r \leftarrow 0] \end{pmatrix} = c_0 \\
(S)^\circ &= \langle \hat{\tau}_\tau \uparrow^* \mid r \uparrow^{-r} \cdot (\bar{\mu}r. (c!)^* \uparrow^r) \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_1)^\circ \uparrow r \uparrow \delta] [r \leftarrow v_0 \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \\
&\rightarrow^* \left\langle \hat{\tau}_\tau \mid v_0 \cdot \left(\bar{\mu}r. (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow \delta] [s \leftarrow (\mathcal{P}_1)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \right) \right\rangle \\
&\rightarrow \left\langle 0 \mid \left(\bar{\mu}r. (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow \delta] [s \leftarrow (\mathcal{P}_1)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \right) \right\rangle \\
&\rightarrow^* (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow \delta \uparrow r] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow \delta \uparrow r] [s \leftarrow (\mathcal{P}_1)^\circ \uparrow \delta \uparrow r] [\delta \leftarrow (s_3)^\circ \uparrow r] [r \leftarrow 0] \end{pmatrix} = s_0
\end{aligned}$$

- Case $v_0 : \tau'$ where $\tau' \neq \tau$: This case is similar to the previous one.

- Case $mustbe_\tau$:

$$\begin{aligned}
S &= \left(mustbe_\tau; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(c!, v_0, s_1, s_2, s_3, \langle s_4, s_5 \rangle \right) = S'
\end{aligned}$$

$$\begin{aligned}
(S')^\circ &= (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) = c_0 \\
(S)^\circ &= \langle \mu_{-} (c!)^* \mid \tilde{\mu}_{-} \langle r \uparrow^{-r} \mid []_A \uparrow^* \rangle \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) \\
&\rightarrow^* \left\langle \mu_{-} \left((c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ] \end{array} \right) \mid \tilde{\mu}_{-} \langle r \uparrow^{-r} \mid []_A \uparrow^* \rangle \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ] \end{array} \right) \right\rangle \\
&\rightarrow^* (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) = c_0
\end{aligned}$$

- **Case push^k; c!:** let $k = \alpha$.

$$\begin{aligned}
S &= \left[\text{push}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right] \\
&\xrightarrow{\text{PPL}} \left[c!, v_0, v_0 \bullet s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right] = S' \\
(S')^\circ &= (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (v_0)^\circ \cdot (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ] \end{array} \right) \rightsquigarrow_{\uparrow} (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta \uparrow \alpha] \\ [r \leftarrow (v_0)^\circ \uparrow \delta \uparrow \alpha] \\ [\delta \leftarrow (s_3)^\circ \uparrow \alpha] \\ [\alpha \leftarrow (v_0)^\circ \cdot (s_1)^\circ] \end{array} \right) = c_0
\end{aligned}$$

$$\begin{aligned}
(S)^\circ &= \langle (\mu\alpha.(c!)^* \uparrow^\alpha \mid (r \uparrow^{-r}) \cdot (\alpha \uparrow^{-\alpha})) \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow^s \uparrow^r \uparrow^\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^\delta] \\ [r \leftarrow (v_0)^\circ \uparrow^\delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) \rangle \\
&\rightarrow^* \left\langle \mu\alpha. \left((c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^s \uparrow^r \uparrow^\delta \uparrow^\alpha] [\gamma \leftarrow (s_2)^\circ \uparrow^s \uparrow^r \uparrow^\delta \uparrow^\alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^\delta \uparrow^\alpha] [r \leftarrow (v_0)^\circ \uparrow^\delta \uparrow^\alpha] [\delta \leftarrow (s_3)^\circ \uparrow^\alpha] \end{array} \right) \right) \mid (v_0)^\circ \cdot (s_1)^\circ \right\rangle \\
&\rightarrow^* (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^s \uparrow^r \uparrow^\delta \uparrow^\alpha] [\gamma \leftarrow (s_2)^\circ \uparrow^s \uparrow^r \uparrow^\delta \uparrow^\alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^\delta \uparrow^\alpha] [r \leftarrow (v_0)^\circ \uparrow^\delta \uparrow^\alpha] [\delta \leftarrow (s_3)^\circ \uparrow^\alpha] [\alpha \leftarrow (v_0)^\circ \cdot (s_1)^\circ] \end{array} \right) \\
&= c_0
\end{aligned}$$

The cases for $k = \gamma$ and $k = \delta$ are similar.

- **Case pop^k; c!:** let $k = \alpha$.

$$\begin{aligned}
S &= \left[\text{pop}^\alpha; c!, v_0, v_1 \bullet s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right] \\
&\xrightarrow{\text{PPL}} \left[c!, v_1, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right] = S'
\end{aligned}$$

$$\begin{aligned}
(S')^\circ &= (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow^s \uparrow^r \uparrow^\delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^\delta] \\ [r \leftarrow (v_1)^\circ \uparrow^\delta] \\ [\delta \leftarrow (s_3)^\circ] \end{array} \right) \sim_{\uparrow (c!)^*} \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^s \uparrow^\delta \uparrow^r \uparrow^\alpha] \\ [\gamma \leftarrow (s_2)^\circ \uparrow^s \uparrow^\delta \uparrow^r \uparrow^\alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^\delta \uparrow^r \uparrow^\alpha] \\ [\delta \leftarrow (s_3)^\circ \uparrow^r \uparrow^\alpha] \\ [r \leftarrow (v_1)^\circ \uparrow^\alpha] \\ [\alpha \leftarrow (s_1)^\circ] \end{array} \right) = c_0 \\
(S)^\circ &= \langle (\lambda r. \mu\alpha.(c!)^* \uparrow^r \uparrow^\alpha \mid \alpha \uparrow^{-\alpha}) \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] [\gamma \leftarrow (s_2)^\circ \uparrow^\alpha \uparrow^s \uparrow^r \uparrow^\delta] \\ [\alpha \leftarrow (v_1)^\circ \cdot (s_1)^\circ \uparrow^s \uparrow^r \uparrow^\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^r \uparrow^\delta] \\ [r \leftarrow (v_0)^\circ \uparrow^\delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) \rangle \\
&\rightarrow^* \left\langle \lambda r. \left(\mu\alpha.(c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow^\gamma \uparrow^s \uparrow^\delta \uparrow^r] [\gamma \leftarrow (s_2)^\circ \uparrow^s \uparrow^\delta \uparrow^r] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow^\delta \uparrow^r] [\delta \leftarrow (s_3)^\circ \uparrow^r] \end{array} \right) \right) \mid (v_1)^\circ \cdot (s_1)^\circ \right\rangle
\end{aligned}$$

$$\begin{aligned}
& \rightarrow^* \left\langle \mu\alpha. \left((c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow s\uparrow\delta\uparrow r\uparrow\alpha] [\gamma \leftarrow (s_2)^\circ \uparrow s\uparrow\delta\uparrow r\uparrow\alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow\delta\uparrow r\uparrow\alpha] [\delta \leftarrow (s_3)^\circ \uparrow r\uparrow\alpha] [r \leftarrow (v_1)^\circ \uparrow\alpha] \end{array} \right) \right) \right\rangle \Big|_{(s_1)^\circ} \\
& \rightarrow^* (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow s\uparrow\delta\uparrow r\uparrow\alpha] [\gamma \leftarrow (s_2)^\circ \uparrow s\uparrow\delta\uparrow r\uparrow\alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow\delta\uparrow r\uparrow\alpha] [\delta \leftarrow (s_3)^\circ \uparrow r\uparrow\alpha] [r \leftarrow (v_1)^\circ \uparrow\alpha] [\alpha \leftarrow (s_1)^\circ] \end{array} \right) \\
& = c_0
\end{aligned}$$

The cases for $k = \gamma$ and $k = \delta$ are similar.

- **Case dump^k; c!:** let $k = \alpha$.

$$\begin{aligned}
S &= \left(\text{dump}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S' \\
(S')^\circ &= (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow\alpha\uparrow s\uparrow r\uparrow\delta] [\gamma \leftarrow (s_2)^\circ \uparrow\alpha\uparrow s\uparrow r\uparrow\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s\uparrow r\uparrow\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r\uparrow\delta] [r \leftarrow (v_0)^\circ \uparrow\delta] [\delta \leftarrow (s_1)^\circ] \end{array} \right) = c_0 \\
(S)^\circ &= \langle (\mu\delta. (c!)^* \uparrow^\delta \mid \alpha \uparrow^{-\alpha}) \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow\alpha\uparrow s\uparrow r\uparrow\delta] [\gamma \leftarrow (s_2)^\circ \uparrow\alpha\uparrow s\uparrow r\uparrow\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s\uparrow r\uparrow\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r\uparrow\delta] \\ [r \leftarrow (v_0)^\circ \uparrow\delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) \right\rangle \\
&\rightarrow^* \left\langle \mu\delta. \left((c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow\alpha\uparrow s\uparrow r\uparrow\delta] [\gamma \leftarrow (s_2)^\circ \uparrow\alpha\uparrow s\uparrow r\uparrow\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s\uparrow r\uparrow\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r\uparrow\delta] [r \leftarrow (v_0)^\circ \uparrow\delta] \end{array} \right) \right) \right\rangle \Big|_{(s_1)^\circ} \\
&\rightarrow^* (c!)^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^* \uparrow\gamma\uparrow\alpha\uparrow s\uparrow r\uparrow\delta] [\gamma \leftarrow (s_2)^\circ \uparrow\alpha\uparrow s\uparrow r\uparrow\delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s\uparrow r\uparrow\delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r\uparrow\delta] [r \leftarrow (v_0)^\circ \uparrow\delta] [\delta \leftarrow (s_1)^\circ] \end{array} \right) \\
&= c_0
\end{aligned}$$

The case for $k = \gamma$ is similar.

- **Case retrieve^k; c!:** let $k = \alpha$.

$$\begin{aligned}
S &= \left(\text{retrieve}^\alpha; c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(c!, v_0, s_3, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S'
\end{aligned}$$

$$\begin{aligned}
(S')^\circ &= (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_3)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] \\ [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \sim_{\uparrow} (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] \\ [\gamma \leftarrow (s_2)^\circ \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta \uparrow \alpha] \\ [r \leftarrow (v_0)^\circ \uparrow \delta \uparrow \alpha] \\ [\delta \leftarrow (s_3)^\circ \uparrow \alpha] \\ [\alpha \leftarrow (s_3)^\circ] \end{pmatrix} = c_0 \\
(S)^\circ &= \langle (\mu\alpha. (c!)^* \uparrow \alpha \mid \delta \uparrow^{-\delta}) \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \rangle \\
&\rightarrow^* \left\langle \mu\alpha. \left((c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] [\gamma \leftarrow (s_2)^\circ \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta \uparrow \alpha] [r \leftarrow (v_0)^\circ \uparrow \delta \uparrow \alpha] [\delta \leftarrow (s_3)^\circ \uparrow \alpha] \end{pmatrix} \right) \right\rangle \Big|_{(s_3)^\circ} \\
&\rightarrow^* (c!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] [\gamma \leftarrow (s_2)^\circ \uparrow s \uparrow r \uparrow \delta \uparrow \alpha] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta \uparrow \alpha] [r \leftarrow (v_0)^\circ \uparrow \delta \uparrow \alpha] [\delta \leftarrow (s_3)^\circ \uparrow \alpha] [\alpha \leftarrow (s_3)^\circ] \end{pmatrix} \\
&= c_0
\end{aligned}$$

The case for $k = \gamma$ is similar.

• **Case do r!:**

$$S = \left(\text{do } r!, @ (v_0), s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right)$$

$$\xrightarrow{\text{PPL}} \left(\mathcal{P}_0[v_0], 0, s_1, s_2, \ominus, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S'$$

$$(S')^\circ = (\mathcal{P}_0[v_0])^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{pmatrix} = c_0$$

$$(S)^\circ = (r \uparrow^{-r} \mid s \uparrow^{-s} \cdot \mathcal{M} \cdot \epsilon \uparrow^*) \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow \widehat{\sigma}_{v_0} \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix}$$

$$\begin{aligned}
& \rightarrow^* \langle \widehat{\sigma}_{v_0} \mid (\mathcal{P}_0)^\circ \cdot (\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \cdot \epsilon \rangle \\
& \rightarrow^* (\mathcal{P}_0[v_0])^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right) \\
& = c_0
\end{aligned}$$

• **Case do @(*v*)!:**

$$\begin{aligned}
S &= \left(\text{do } @(\mathit{v})!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(\mathcal{P}_0[v], 0, s_1, s_2, \odot, \langle \mathcal{P}_0, \Phi_0 \rangle \right) = S' \\
(S')^\circ &= (\mathcal{P}_0[v])^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right) = c_0 \\
(S)^\circ &= \langle \widehat{\sigma}_v \uparrow^* \mid s \uparrow^{-s} \cdot \mathcal{M} \cdot \epsilon \uparrow^* \rangle \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right) \\
&\rightarrow^* \langle \widehat{\sigma}_{v_0} \mid (\mathcal{P}_0)^\circ \cdot (\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \cdot \epsilon \rangle \\
&\rightarrow^* \langle (\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \mid (\mathcal{P}_0[v_0])^\circ \rangle \\
&= \langle (\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \mid \varphi \lambda. \gamma \lambda. \alpha \lambda. \bar{\mu} s. \langle \lambda r. \mu \delta. (\mathcal{P}_0[v_0])^* \mid (0 \cdot \epsilon) \uparrow^* \rangle \rangle \\
&\rightarrow^* (\mathcal{P}_0[v_0])^* \left(\begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right) \\
&= c_0
\end{aligned}$$

• **Case do @(*n*)?@(*m*)!:** This case follows the same lines as do @(*x*)*n*

• **Case call (\mathcal{P}_1); *c*!:**

$$\begin{aligned}
S &= \left(\text{call}(\mathcal{P}_1); c!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right) \\
&\xrightarrow{\text{PPL}} \left(\mathcal{P}_1[1], 0, \odot, s_3, \odot, \left\langle \mathcal{P}_1, \left[c!, 0, s_1, s_2, \odot, \langle \mathcal{P}_0, \Phi_0 \rangle \right] \right\rangle \right) = S'
\end{aligned}$$

$$\begin{aligned}
(S')^\circ &= (\mathcal{P}_1[1])^* \left(\begin{array}{l} [\varphi \leftarrow (c!, 0, s_1, s_2, \odot, \langle \mathcal{P}_0, \Phi_0 \rangle) \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_3)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow \epsilon \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_1)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right) = c_0 \\
(S)^\circ &= \left\langle \begin{array}{l} (\mathcal{M} \cdot \overline{\mathcal{M}^r}((c!)^*) \uparrow^*) \cdot \epsilon \uparrow^* \cdot \delta \uparrow^{-\delta} \cdot \epsilon \uparrow^* \\ \cdot (\mathcal{P})^\circ \uparrow^* \mid \overline{\mathcal{M}}((\text{do } @ (1)!)^*) \uparrow^{\varphi \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta} \end{array} \right\rangle \begin{array}{l} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\alpha \leftarrow (s_1)^\circ \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_0)^\circ \uparrow r \uparrow \delta] \\ [r \leftarrow (v_0)^\circ \uparrow \delta] [\delta \leftarrow (s_3)^\circ] \end{array} \\
&\rightarrow^* \left\langle \begin{array}{l} ((\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \cdot \overline{\mathcal{M}^r}((c!)^*) \cdot \epsilon) \cdot (s_3)^\circ \cdot \epsilon \cdot (\mathcal{P}_1)^\circ \\ \mid \varphi \lambda. \gamma \lambda. \alpha \lambda. \tilde{\mu} s. \langle \lambda r. \mu \delta. (\text{do } @ (1)!)^* \mid (0 \cdot \epsilon) \uparrow^{\varphi \uparrow \gamma \uparrow \alpha \uparrow s} \rangle \end{array} \right\rangle \\
&\rightarrow^* \langle \lambda r. \mu \delta. (\text{do } @ (1)!)^* \mid (0 \cdot \epsilon) \uparrow^{\varphi \uparrow \gamma \uparrow \alpha \uparrow s} \rangle \\
&\quad [\varphi \leftarrow ((\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \cdot \overline{\mathcal{M}^r}((c!)^*) \cdot \epsilon) \uparrow \gamma \uparrow \alpha \uparrow s] \\
&\quad [\gamma \leftarrow (s_3)^\circ \uparrow \alpha \uparrow s] [\alpha \leftarrow \epsilon \uparrow s] [s \leftarrow (\mathcal{P}_1)^\circ] \\
&\rightarrow^* (\text{do } @ (1)!)^* \left(\begin{array}{l} [\varphi \leftarrow ((\Phi_0)^\circ \cdot (s_2)^\circ \cdot (s_1)^\circ \cdot (\mathcal{P}_0)^\circ \cdot \overline{\mathcal{M}^r}((c!)^*) \cdot \epsilon) \uparrow \gamma \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] \\ [\gamma \leftarrow (s_3)^\circ \uparrow \alpha \uparrow s \uparrow r \uparrow \delta] [\alpha \leftarrow \epsilon \uparrow s \uparrow r \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_1)^\circ \uparrow r \uparrow \delta] [r \leftarrow 0 \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right)
\end{aligned}$$

By the sequence shown in the case for $\text{do } @ (a)!$, this reduces to c_0

• **Case restore!:**

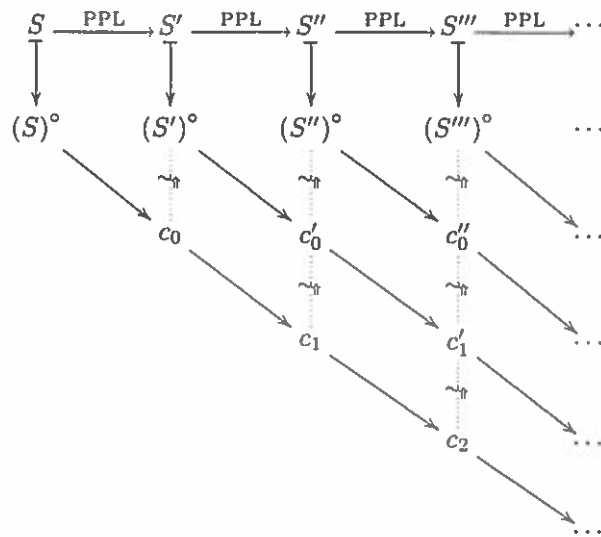
$$\begin{aligned}
\text{let } \Phi_1 &= \left[c_1!, v_1, s_4, s_5, s_6, \langle \mathcal{P}_2, \Phi_0 \rangle \right]. \\
S &= \left[\text{restore!}, v_0, s_1, s_2, s_3, \langle \mathcal{P}_1, \Phi_1 \rangle \right] \\
&\xrightarrow{\text{PPL}} \left[c_1!, v_0, s_4, s_5, \odot, \langle \mathcal{P}_2, \Phi_0 \rangle \right] = S'
\end{aligned}$$

$$\begin{aligned}
(S')^\circ &= (c_1!)^* \begin{pmatrix} [\varphi \leftarrow (\Phi_0)^\circ \uparrow \uparrow \alpha \uparrow \uparrow \delta] \\ [\gamma \leftarrow (s_5)^\circ \uparrow \alpha \uparrow \uparrow \delta] \\ [\alpha \leftarrow (s_4)^\circ \uparrow \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_2)^\circ \uparrow \uparrow \delta] \\ [r \leftarrow v_0 \uparrow^\delta] \\ [\delta \leftarrow \epsilon] \end{pmatrix} \sim_{\uparrow (c_1!)^*} \begin{pmatrix} [r \leftarrow v_0 \uparrow \varphi \uparrow \alpha \uparrow \uparrow \delta] \\ [\varphi \leftarrow (\Phi_0)^\circ \uparrow \uparrow \alpha \uparrow \uparrow \delta] \\ [\gamma \leftarrow (s_5)^\circ \uparrow \alpha \uparrow \uparrow \delta] \\ [\alpha \leftarrow (s_4)^\circ \uparrow \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_2)^\circ \uparrow^\delta] \\ [\delta \leftarrow \epsilon] \end{pmatrix} = c_0 \\
(S)^\circ &= \left\langle \begin{array}{l} (\lambda s. \lambda t. \mu. \langle t \uparrow^* \mid r \uparrow^{-r} \uparrow^t \\ \cdot s \uparrow^{-s} \uparrow^t \cdot \epsilon \uparrow^* \uparrow^t \rangle) \uparrow^s \mid \varphi \uparrow^{-\varphi} \end{array} \right\rangle \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \uparrow \alpha \uparrow \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow \uparrow \delta] \\ [s \leftarrow (\mathcal{P}_1)^\circ \uparrow \uparrow \delta] [r \leftarrow v_0 \uparrow^\delta] [\delta \leftarrow (s_3)^\circ] \end{pmatrix} \\
&\rightarrow^* \left\langle \begin{array}{l} \lambda s. \lambda t. \mu. \langle (t \uparrow^* \mid r \uparrow^{-r} \uparrow^t \cdot s \uparrow^{-s} \uparrow^t \cdot \epsilon \uparrow^* \uparrow^t) [\varphi \leftarrow (\Phi_1)^\circ \uparrow \uparrow \alpha \uparrow \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow \uparrow \delta] [r \leftarrow v_0 \uparrow^\delta] [\delta \leftarrow (s_3)^\circ] \end{array} \right\rangle \\
&\quad \mid (\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ \cdot \overline{\mathcal{M}^r}((c_1!)^*) \\
&\rightarrow^* \langle t \uparrow^* \mid r \uparrow^{-r} \uparrow^t \cdot s \uparrow^{-s} \uparrow^t \cdot \epsilon \uparrow^* \uparrow^t \rangle \begin{pmatrix} [\varphi \leftarrow (\Phi_1)^\circ \uparrow \uparrow \alpha \uparrow \uparrow \delta] \\ [\gamma \leftarrow (s_2)^\circ \uparrow \alpha \uparrow \uparrow \delta] [\alpha \leftarrow (s_1)^\circ \uparrow \uparrow \delta] \\ [r \leftarrow v_0 \uparrow^\delta] [\delta \leftarrow (s_3)^\circ] \\ [s \leftarrow (\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ \uparrow^t] \\ [t \leftarrow \overline{\mathcal{M}^r}((c_1!)^*)] \end{pmatrix} \\
&\rightarrow^* \langle \overline{\mathcal{M}^r}((c_1!)^*) \mid v_0 \cdot ((\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ) \cdot \epsilon \rangle \\
&= \left\langle \begin{array}{l} \lambda r. \lambda s. \mu. \langle s \uparrow^r \mid (\varphi \lambda. \gamma \lambda. \alpha \lambda. \tilde{\mu} s. \langle \mu \delta. (c_1!)^* \mid \epsilon \uparrow^{-\delta} \rangle) \uparrow^s \rangle \\ \mid v_0 \cdot ((\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ) \cdot \epsilon \end{array} \right\rangle \\
&\rightarrow^* \langle s \uparrow^r \mid (\varphi \lambda. \gamma \lambda. \alpha \lambda. \tilde{\mu} s. \langle \mu \delta. (c_1!)^* \mid \epsilon \uparrow^{-\delta} \rangle) \uparrow^s \rangle \begin{pmatrix} [r \leftarrow v_0 \uparrow^s] \\ [s \leftarrow (\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ] \end{pmatrix} \\
&\rightarrow^* \langle ((\Phi_0)^\circ \cdot (s_5)^\circ \cdot (s_4)^\circ \cdot (\mathcal{P}_2)^\circ \mid \varphi \lambda. \gamma \lambda. \alpha \lambda. \tilde{\mu} s. \langle \mu \delta. (c_1!)^* \mid \epsilon \uparrow^{-\delta} \rangle [r \leftarrow v_0 \uparrow \varphi \uparrow \alpha \uparrow \uparrow \delta]) \rangle
\end{aligned}$$

$$\begin{aligned}
& \rightarrow^* \langle \mu\delta. (c_1!)^* \mid \epsilon \uparrow^{-\delta} \rangle \left(\begin{array}{l} [r \leftarrow v_0 \uparrow \varphi \uparrow \gamma \uparrow \alpha \uparrow s] [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s] \\ [\gamma \leftarrow (s_5)^\circ \uparrow \alpha \uparrow s] [\alpha \leftarrow (s_4)^\circ \uparrow s] [s \leftarrow (\mathcal{P}_2)^\circ] \end{array} \right) \\
& \rightarrow^* (c_1!)^* \left(\begin{array}{l} [r \leftarrow v_0 \uparrow \varphi \uparrow \gamma \uparrow \alpha \uparrow s \uparrow \delta] [\varphi \leftarrow (\Phi_0)^\circ \uparrow \gamma \uparrow \alpha \uparrow s \uparrow \delta] \\ [\gamma \leftarrow (s_5)^\circ \uparrow \alpha \uparrow s \uparrow \delta] [\alpha \leftarrow (s_4)^\circ \uparrow s \uparrow \delta] [s \leftarrow (\mathcal{P}_2)^\circ \uparrow \delta] [\delta \leftarrow \epsilon] \end{array} \right) \\
& = c_0
\end{aligned}$$

■

Taken with Lemma II.3.6 and Theorem II.3.4, Theorem II.3.5 shows the following diagram:



By strategically ignoring some of the nodes, we can replace the $\xrightarrow{\text{PPL}}$ reductions with $\xrightarrow{\text{PPLI}}$. However, to obtain the result we want, we need to show in the modified diagram that $c_0 = (S')^\circ$. Although Theorem II.3.4 is the best we can do for arbitrary $\lambda\mu\bar{\mu}\uparrow$ commands, we are not dealing with arbitrary commands. Since each PPL do command rebinds all of the variables, the translation of each do command will result in a canonical ordering of the outer substitutions.

Lemma II.3.7 *If $S \xrightarrow{\text{PPL}} S'$ and*

- $S = \left(\text{do } r!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right)$
- *or* $S = \left(\text{do } @(v)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right)$
- *or* $S = \left(\text{do } @(n)?@(m)!, v_0, s_1, s_2, s_3, \langle \mathcal{P}_0, \Phi_0 \rangle \right)$

then $(S)^\circ \rightarrow^ (S')^\circ$*

Proof. Each case is shown in the proof of Theorem II.3.5.

Theorem II.3.8 *For any PPL state S , if $S \xrightarrow{\text{PPL!}} S'$, then $(S)^\circ \rightarrow^* (S')^\circ$*

Proof. This is immediate from Lemma II.3.7 and the observation from definition I.3.1 that $\xrightarrow{\text{PPL!}}$ always ends by reducing a `do` instruction.

CONCLUSION

This thesis has demonstrated the ability of sequent calculi to represent actual machines. This extends the previous work of Bohannon (2004) in simulating the SECD and other abstract machines. Although the JVM is formally abstract, in the sense that its instructions do not operate on any actual hardware, it is clearly a fully featured computing environment. Our work, therefore, has shown the feasibility of developing a calculus in sequent form that can model the low-level details of a functioning machine. The calculus used here is based on the calculus of Curien and Herbelin (2000).

In creating an intermediate abstract machine, we maintained the constant time nature of the instruction execution steps. These intermediate instructions can be seen as microcode in some implementation of the JVM. The stack-based intermediate machine allows us to translate its instructions into the list-based term and context expressions of the $\lambda\mu\tilde{\mu}\dagger$ calculus. By using immediate substitution, we were again able to maintain the constant time nature of each computational step.

Our approach was to combine the function of the JVM verifier and interpreter into a single target semantics. This strategy produces a result in the calculus that at once simulates the JVM interpreter reductions, and also produces terms that type-check exactly when the verifier passes on the JVML instructions. However, to fully realize this goal, our work would have to be extended to encompass a representation of the heap. In particular, our work does not check the proper initialization of references. Therefore, our system is not as powerful as Freund and Mitchell (1998).

This work shows the feasibility of developing a verifier based on the logic of an off-the-shelf calculus. With such a verifier, neither the rules of the translation, nor the

rules of the logic need be a part of the trusted code base. This allows a much more secure verifier than one based on control flow. Our work has shown how to accomplish this using a sequent calculus that is much closer to the machine execution than other proposed calculi.

BIBLIOGRAPHY

- Bohannon, Aaron. *Sequent Calculus and Abstract Machines*. Master's thesis, University of Oregon, 2004.
- Curien, Pierre-Louis, and Hugo Herbelin. "The duality of computation." In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2000, 233–243.
- Freund, Stephen N., and John C. Mitchell. "A type system for object initialization in the Java bytecode language." In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 1998, 310–327.
- Higuchi, Tomoyuki, and Atsushi Ohori. "Java bytecode as a typed term calculus." In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM Press, 2002, 201–211. <http://www.jaist.ac.jp/ohori/research/jvmcalc.ps>.
- Lindholm, Tim, and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999, second edition.
- Qian, Zhenyu. "A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines." In *Formal Syntax and Semantics of Java*. London, UK: Springer-Verlag, 1999, 271–312.
- Stata, Raymie, and Martin Abadi. "A type system for Java bytecode subroutines." *ACM Trans. Program. Lang. Syst.* 21, 1: (1999) 90–137.