

SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis

Eduardo Bonelli

Adriana Compagnoni

Ricardo Medel

{ebonelli, abc, rmedel}@cs.stevens-tech.edu

Stevens Institute of Technology

Abstract

We study information flow for a typed assembly language where security types restrict information flow. Inspired by recent work in continuation-based information flow analysis, our language, Secure Information Flow TAL (SIFTAL), uses low-level linear continuations in order to impose a stack discipline on the control flow of programs.

The challenge posed by studying information flow analysis at the assembly language level is many-fold. On the one hand, the well behaved control constructs of high-level languages are not available, and, on the other hand, the role of an unbounded number of variables is played by a finite number of registers that need to be reused not only with different types, but also with different security levels.

Non-interference refers to the desirable property of systems of multilevel security architecture that states that information stored at a high security level does not affect computed low security level values. Our main contributions are a type system for checking that typed assembly language programs enjoy non-interference and its proof of soundness.

Furthermore, SIFTAL is the first typed assembly language with security types for information flow analysis, and our proof is the first proof of non-interference for a MIPS-style typed assembly language.

1. Introduction

The confidentiality of information handled by computing systems is of paramount importance in voting, electronic commerce, and in military, medical, and financial applications, just to name a few relevant areas. However, standard mechanisms such as perimeter security in the form of access control or digital signatures for authentication and encryption fail to address the enforcement of information-flow policies. In this scenario, language-based strategies for security offer a promising approach to information flow security. In this paper, we study confidentiality for an assembly language using a language-based approach to security via type-theory.

In a multilevel security architecture information can range from having low to high security level, where low means public and high means confidential. Furthermore, following Denning's work [13], the security levels can be represented as a lattice. Information flow analysis studies whether an attacker can obtain information about the input data by observing the output of the system. In other words, it studies whether low-security computation may be affected by high-level data.

The notion of non-interference states that the computation of low-level values cannot be affected by high-level data, which implies that two executions of the same program, where only the high-level inputs differ in both executions, should not exhibit any observable difference in the program's output.

In this paper we define *SIFTAL*, a typed assembly language for secure information flow analysis with security types. Among its non-standard features, *SIFTAL* has a stack of linear continuations that indicate the program points where different branches of code converge. Our development culminates with a proof that well-typed programs satisfy the non-interference property.

The study of information leaking through covert channels is outside the scope of this work.

1.1. Assembly Languages And Information Flow

High-Level Control Flow. In information flow analysis, a security level is associated with each program execution point—often called **pc**—. This security level is used to detect implicit information flow from high-level values to low-level values. Moreover, control flow analysis is crucial in allowing this security level to decrease where there is no risk of illicit flow of information. Consider the following example where x has high security level and b has low security level.

```
pc = low    if x
pc = high   then  a := 1
pc = high   else  a := 2
pc = low    b := 3
```

Notice that a cannot have low security level, since x can be retrieved from a , violating the non-interference property,

as described above. Entering the branches of the *if-then-else* changes the **pc** to high, indicating that only high-level values can be updated. However, since $b := 3$ is executed after both branches, there is no leakage of information from either a or x to b . Therefore, the **pc** can be changed to low. A standard compilation to assembly language may produce the following code:

```

L1 :  bnz r1, L2    %   if x ≠ 0 goto L2
      mov r2, 1     %   a:=1
      jmp L3
L2 :  mov r2, 2     %   a:=2
L3 :  mov r3, 3     %   b:=3

```

However, we no longer have the block structure of *if-then-else* to be able to lower **pc**, and since, in general, it is undecidable if a sequence of instructions will be executed, we do not have enough information in the assembly code to lower **pc**. Inspired by [27]¹, we solve this problem by including in *SIFTAL* a stack of linear continuations with `cpush` and `jmpcc` to simulate the block structure of high-level control structures such as *if-then-else*. The *SIFTAL* code for this example can be found in Figure 1, where the block at *START* pushes *L3* onto the linear continuation stack. The block at *L3* corresponds to the code following *if-then-else* in the source code. Observe that the block at *L3* can only be executed once, because `jmpcc` at *L1* (*then* branch) or *L2* (*else* branch) removes the top of the continuation stack and jumps to it.

Register Reuse. In an assembly language, information can be stored in a memory location or contained in a register, and while recycling unreachable memory locations is often the job of a garbage collector, memory registers are reused freely by a compiler during the register allocation phase. Using vital liveness analysis information, a compiler will typically use the same register for different variables whose live ranges do not overlap. But in the presence of security levels, the same register can be used for different variables with different security levels, and this presents the first problem. Without the liveness information from the source code, in general, we cannot say just by looking at the assembly code, if a register is being used for one or more variables in the source code.

Consider the following assembly code:

```

1 :  mov r1, 0
2 :  mov r2, 1
3 :  add r2 ← r1 + 0

```

It could be the result of compiling the following non-interferent source code:

```

letlin k = (λ⟨⟩. b:=3; halt ⟨⟩) in if x then {
a:=1; k ⟨⟩} else { a:=2; k ⟨⟩}

```

¹The compilation into the Secure CPS Calculus [27] is:

$$\begin{aligned}
z^h &:= 0^h; \\
x^l &:= 1^l; \\
y^h &:= z^h
\end{aligned}$$

Here r_2 is first used for a low-level variable x^l and then for a high-level variable y^h . Since the live ranges of both variables are non-overlapping this is an acceptable compilation. However, observe that in order to prevent high-level data from affecting low-level outputs, we need to check that in 2 the security level of the value 1 preserves the security level of r_2 , and that in 3, the security level of r_1 preserves the security level of r_2 . In 2, r_2 has security level low, because x^l is low, and since r_1 has security level high from z^h , the assignment in 3 appears to be a violation of the non-interference property, even if it is the compilation of a non-interferent source program.

Observe that we are not concerned about the fact that an attacker could observe that r_2 contains high-level data, because the attacker cannot access the contents of r_2 . As we said before, the leaking of information through covert channels is outside the scope of this work.

The reuse of registers masks the fact that the original two variables had different security levels. However, our intention is to be able to apply our information flow analysis to code where the standard compilation practice of reusing registers freely is allowed.

To do that we need to keep during typechecking the liveness information that allowed the reuse of registers during register allocation. For that we keep a table that for each instruction in the assembly code, indicates what variable is implemented by each register in the instruction. The security information associated with every register will depend on which variable it is implementing. Thus, when the register is reused, the old security level will be ignored because it was associated with the implementation of a different variable. Returning to our example above, line 3 will not invalidate the non-interference property. The table for our example is:

```

1:  (r1, zh)
2:  (r2, xl)
3:  (r2, yh)   and   (r1, zh)

```

We can argue informally that instead of keeping the original code and a table with liveness information, it is equivalent to typecheck an annotated assembly code where reuses of registers names are disambiguated through renaming, for example by adding a suffix with the name of the originating variable in the source code so as to disambiguate reuses of registers. In that case, two occurrences of the same register for different variables will have different names. Similarly, internal variables created during compilation will need to be given distinct names. Then our example becomes:

```

1:  mov rz1, 0
2:  mov rx2, 1
3:  add ry2 ← rz1 + 0

```

```

L1 :      ((⊥){r1 : int⊤, r2 : int⊤, r3 : int⊤} | κl · κl' · ε → {})⊥
        bnz r1, L2;      %   if x ≠ 0 goto L2
        mov r2, 1⊤;      %   a:=1
        jmpcc             %

L2 :      ((⊥){r2 : int⊤, r3 : int⊤} | κl · κl' · ε → {})⊥
        mov r2, 2⊤;      %   a:=2
        jmpcc             %

L3 :      ((⊥){r3 : int⊤} | κl' · ε → {})⊥
        mov r3, 3⊥;      %   b:=3
        jmpcc             %

START :   ((⊥){r1 : int⊤, r2 : int⊤, r3 : int⊤} | ε → {})⊥
        cpush HALT;      %   set default exit continuation to HALT
        cpush L3;        %   set linear continuation to L3
        mov r1, ?⊤;      %   ?⊤:int⊤
        jmp L1           %

HALT :    ((⊥){} | ε → {})⊥
        where κl' = ((⊥){} | ε → {})⊥ and κl = ((⊥){} | κl' · ε → {})⊥

```

Figure 1. Sample code in SIFTAL

where there is no confusion between rx_2 and ry_2 .

In this paper we assume a pre-processing phase that, using liveness information from the source code [1, 2], produces annotated assembly code where register reuse has been disambiguated.

1.2. Related Work

The present work is in the framework of Necula’s Proof Carrying Code [20], where the property of interest is non-interference. Furthermore, typed assembly languages have been advocated for proof-carrying code, and have been an active subject of study for several years now. Experimentation with different ideas is still taking place. Contributions include TAL [19, 12], STAL [18], DTAL [26], Alias Types [23, 25], HBAL [3], and LTAL [11].

Information flow analysis has been an active research area in the past three decades. See [22] for a comprehensive survey. Bell and LaPadula [6] defined an abstract model intended to control information flow, where objects and subjects have a security level (top secret, secret, classified, unclassified). A subject cannot read objects of level higher than its level, and it cannot write objects at levels lower than its level. [15] defines multilevel security policies where information is classified not only by its security level but by its category. Then a subject cannot read information at lower levels on different categories. [13] generalized security levels to lattices. [21, 8] define integrity using a lattice of labels

to prevent bad data to affect good data. [14] studies static analysis of information flow using dataflow analysis. It describes an inference algorithm to determine whether variables have high or low security level.

The notion of *non-interference* was first introduced by Goguen and Meseguer [16, 17], and there has been a significant amount of research on type systems for confidentiality for high-level languages and for models of computation starting with seminal work by Volpano, Smith and Irvine [24]. However, as Sabelfeld and Myers observe [22], there has been hardly any work on information flow analysis for low-level languages.

In [27], Zdancewicz and Myers present a low-level, secure calculus with higher-order, imperative features, and linear continuations. The language has a block structure and binding constructs that *SIFTAL* does not have, and their language does not have registers, only a heap.

In [4], Barthe, Basu, and Rezk define a low-level language with a heap and an operand stack, where instructions load and store transfer values between the top of the stack and the heap, but their language does not have registers that can be reused to store values of different types and different security levels, unlike *SIFTAL*. They also assume that program points reflecting the block structure of high-level constructs are given, instead of being computed with a stack of linear continuations. Barthe and Rezk, in [5], extend the previous work with objects to study information flow for the JVM. In [9], a method for detecting illicit flows for a

sequential fragment of the JVM is presented, and in [7], the method is extended to a subset of the JVM including jumps and subroutines.

2. Syntax of SIFTAL

We assume given a lattice \mathcal{L}_{sec} of *security labels*. The least and greatest elements of this lattice are \perp and \top , respectively. We use \sqsubseteq for the lattice ordering and \sqcup and \sqcap for the lattice join and meet operations, respectively. Security labels are used to assign security levels to integer constants, the program counter, and security types.

The syntax of *SIFTAL* is as follows:

security labels	$l, \mathbf{pc} \in \mathcal{L}_{\text{sec}}$
program	$\Sigma ::= (H, R, B, \mathbf{pc}, C)$
code blocks	$B ::= \text{halt} \mid \text{jmp } v \mid \text{jmpcc} \mid \iota; B$
instructions	$\iota ::= aop \ r_d, r_s, v \mid \text{bnz } r, v \mid \text{mov } r, v \mid \text{ld } r_d, r_s[i] \mid \text{st } r_d[i], r_s \mid \text{cpush } \underline{L}$
arithmetic ops	$aop ::= \text{add} \mid \text{sub} \mid \text{mul}$
operands	$v ::= r \mid w$
word values	$w ::= i^l \mid p \mid L$
heap addresses	$\ell ::= p \mid L \mid \underline{L}$
heap values	$h ::= \langle w, \dots, w \rangle \mid B$
continuation stack	$C ::= \epsilon \mid \underline{L} \cdot C$

We use r, r_i, r_d, r_s, \dots to range over register names. A *SIFTAL* program is a machine configuration $(H, R, B, \mathbf{pc}, C)$ where $H(\text{eap})$ is a partial map from the set of *heap addresses* to the set of *heap values*; $R(\text{egister bank})$ is a partial map from registers names to word values, $B(\text{lock})$ is a code block corresponding to the current program counter; \mathbf{pc} is a security label indicating the security level of the program counter; and $C(\text{ontinuation stack})$ is the linear continuations stack. There are three kinds of heap addresses (or labels): *tuple* labels (p) pointing to heap locations containing tuples of word values, *non-linear* code labels (L) pointing to heap locations containing code blocks, and *linear* code labels (\underline{L}) pointing to heap locations containing a linear continuation. H maps tuple labels to tuple values and code labels to code blocks.

A code block is a sequence of instructions ending in a jump to a code label, a jump to the current linear continuation, or a `halt` instruction. The instruction `jmpcc` forces the program counter to jump to the current continuation. The latter is retrieved from the top of the stack C of pending continuations. New continuations are added to this stack with `cpush`. We sometimes refer to `jmpcc` as a *linear jump* since the continuation to which it refers is handled in a linear way by the type system. *SIFTAL* also has standard

assembly language instructions such as arithmetic operations, conditional branching (`bnz` r, v), move (`mov` r, v), load (`ld` $r_d, r_s[i]$), and store (`st` $r_d[i], r_s$).

2.1. Type Expressions

The type expressions of *SIFTAL* are as follows:

types	$\tau ::= \text{int} \mid \kappa \mid \langle \rho_1, \dots, \rho_n \rangle$
security types	$\rho, \sigma ::= \tau^l$
block types	$\kappa ::= \langle \mathbf{pc} \rangle \Xi \rightarrow \{ \}$
block contexts	$\Xi ::= \Gamma \mid \Delta$
register bank types	$\Gamma ::= \{ r_1 : \sigma_1, \dots, r_n : \sigma_n \}$
contin. stack types	$\Delta ::= \epsilon \mid \kappa^l \cdot \Delta$
heap types	$\Psi ::= \{ \ell_1 : \sigma_1, \dots, \ell_n : \sigma_n \}$
program types	$\Omega ::= [\Psi, \Gamma, \Delta]$

int is the type of integer constants; κ is the type of code labels and hence continuations; and $\langle \rho_0, \dots, \rho_{n-1} \rangle$ is the type of tuple labels. *Security types* are types annotated with a security label (τ^l).

Register bank types (Γ) map registers to types. A *continuation stack type* (Δ) is a stack of block types each of which is annotated with a security label, and ϵ is the type of the empty stack. We assume that there is a dedicated register `csp` containing a continuation stack pointer, and that Δ is the type assigned to the contents of this register. Moreover, we assume that if $\Gamma = \{ r_1 : \sigma_1, \dots, r_n : \sigma_n \}$, then $r_i \neq \text{csp}$ for all $i \in \{1..n\}$. The intuition for a *block type* ($\langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{ \}$) is that it contains the security label of the program counter (\mathbf{pc}), the type of the registers in the code block (Γ), and the type of the stack of linear continuations (Δ) and “ $\{ \}$ ” indicates that the block does not return a value since it ends with a jump or `halt`. A *heap type* (Ψ) is a mapping from heap addresses to security types.

If $\Gamma = \{ r_1 : \sigma_1, \dots, r_n : \sigma_n \}$, then $\text{Dom}(\Gamma) = \{ r_1, \dots, r_n \}$. We write $\Gamma[r := \sigma]$ for the register bank type resulting from updating Γ with $r : \sigma$. We define $\text{label}(\tau^l) = l$ and the function \hat{R} as $\hat{R}(v) = R(r)$ if $v = r$ and $\hat{R}(v) = w$ if $v = w$. The function $\Psi \cup \Gamma$ is defined as follows: $(\Psi \cup \Gamma)(v) = \Gamma(r)$, if $v = r$; $(\Psi \cup \Gamma)(v) = \Psi(v)$, if $v = p$ or $v = L$, and $(\Psi \cup \Gamma)(v) = \text{int}^l$, if $v = i^l$.

2.2. Operational Semantics

The operational semantics is defined by the transition relation given in Figure 2. A typed machine configuration $\Sigma : \Omega$ is said to reduce to $\Sigma' : \Omega'$, if $\Sigma : \Omega \mapsto \Sigma' : \Omega'$. We use \mapsto^* for the reflexive-transitive closure of \mapsto . As already discussed, \mathbf{pc} represents the level of information that could be learned at the particular program point. Consider for example, the `OS_Bnz1` rule. The fact that no jump takes place is evidence that the value of the register r is zero, therefore the level of information that can be learned must be updated

$$\begin{array}{c}
\frac{\hat{R}(v) = L \quad H(L) = B \quad (\Psi \cup \Gamma)(v) = (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l}{(H, R, \text{jmp } v, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R, B, \mathbf{pc} \sqcup \mathbf{pc}' \sqcup l, C) : [\Psi, \Gamma, \Delta]} \text{OS_Jump} \\
\\
\frac{H(\underline{L}) = B \quad \kappa = \langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\}}{(H, R, \text{jmpcc}, \mathbf{pc}, \underline{L} \cdot C) : [\Psi, \Gamma, \kappa^l \cdot \Delta] \mapsto (H, R, B, \mathbf{pc}' \sqcup l, C) : [\Psi, \Gamma, \Delta]} \text{OS_Jumpcc} \\
\\
\frac{n = \hat{R}(v) \oplus \hat{R}(r_s) \quad l = \text{label}((\Psi \cup \Gamma)(v)) \sqcup \text{label}(\Gamma(r_s)) \quad \mathbf{pc} \sqsubseteq l \quad l \sqsubseteq \text{label}(\Gamma(r_d))}{(H, R, \text{aop } r_d, r_s, v; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R[r_d := n^l], B, \mathbf{pc}, C) : [\Psi, \Gamma[r_d := \text{int}^l], \Delta]} \text{OS_Arith} \\
\\
\frac{\Gamma(r) = \text{int}^l \quad \hat{R}(r) = 0}{(H, R, \text{bnz } r, v; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R, B, \mathbf{pc} \sqcup l, C) : [\Psi, \Gamma, \Delta]} \text{OS_Bnz1} \\
\\
\frac{\Gamma(r) = \text{int}^{l_1} \quad \hat{R}(r) \neq 0 \quad (\Psi \cup \Gamma)(v) = (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^{l_2} \quad H(\hat{R}(v)) = B'}{(H, R, \text{bnz } r, v; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R, B', \mathbf{pc} \sqcup \mathbf{pc}' \sqcup l_1 \sqcup l_2, C) : [\Psi, \Gamma, \Delta]} \text{OS_Bnz2} \\
\\
\frac{\mathbf{pc} \sqsubseteq \text{label}((\Psi \cup \Gamma)(v)) \quad \text{label}((\Psi \cup \Gamma)(v)) \sqsubseteq \text{label}(\Gamma(r_d))}{(H, R, \text{mov } r_d, v; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R[r_d := \hat{R}(v)], B, \mathbf{pc}, C) : [\Psi, \Gamma[r_d := (\Psi \cup \Gamma)(v)], \Delta]} \text{OS_Mov} \\
\\
\frac{\Gamma(r_s) = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \quad \hat{R}(r_s) = p \quad \text{label}(\sigma_i) \sqsubseteq \text{label}(\Gamma(r_d)) \quad H(p) = \langle w_0, \dots, w_i, \dots, w_{n-1} \rangle \quad \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i)}{(H, R, \text{ld } r_d, r_s[i]; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R[r_d := w_i], B, \mathbf{pc}, C) : [\Psi, \Gamma[r_d := \sigma_i], \Delta]} \text{OS_Load} \\
\\
\frac{\hat{R}(r_d) = p \quad H(p) = \langle w_0, \dots, w_i, \dots, w_{n-1} \rangle \quad \Gamma(r_d) = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \quad \Gamma(r_s) = \sigma_i \quad \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i)}{(H, R, \text{st } r_d[i], r_s; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H[p := \langle w_0, \dots, \hat{R}(r_s), \dots, w_{n-1} \rangle], R, B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta]} \text{OS_Store} \\
\\
\frac{\mathbf{pc} \sqsubseteq \text{label}(\Psi(\underline{L}))}{(H, R, \text{cpush } \underline{L}; B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R, B, \mathbf{pc}, \underline{L} \cdot C) : [\Psi, \Gamma, \Psi(\underline{L}) \cdot \Delta]} \text{OS_Push}
\end{array}$$

Figure 2. Operational semantics

to include the security level of r . Likewise, in the case of the OS_Jmp rule the new \mathbf{pc} must be joined with that of the destination label and the security type in order to avoid implicit flows. Note, however, that in the case of OS_Jmpcc the \mathbf{pc} may potentially be lowered (to some previous level indicated by the type of the top continuation on the stack).

The rule OS_Push models the addition of a new linear continuation to the stack. The security level of the continuation must be at least that of the current \mathbf{pc} . This avoids the situation in which a continuation is called through jmpcc and a security level lower than the one in which the original cpush took place is adopted.

2.3. Typing Rules

In order for a program to be well-typed with type $[\Psi, \Gamma, \Delta]$, each of its components must be considered. The heap must be well-typed with type Ψ , the register bank must be well-typed with type Γ , the current code block must be well-typed with type $\langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{\}$ and the continuation stack must be well-typed with type Δ .

$$\begin{array}{c}
\frac{\triangleright H : \Psi \text{ heap} \quad \triangleright_{\Psi} R : \Gamma \text{ regFile} \quad \Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk} \quad \triangleright_{\Psi} C : \Delta \text{ cstack}}{\triangleright (H, R, B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \text{ program}} \text{T_Program} \\
\\
\frac{\text{Dom}(H) = \text{Dom}(\Psi) \quad \forall \ell \in \text{Dom}(H). \triangleright_{\Psi} H(\ell) : \Psi(\ell) \text{ hval}}{\triangleright H : \Psi \text{ heap}} \text{T_Heap} \\
\\
\frac{\forall r \in \text{Dom}(\Gamma). \triangleright_{\Psi} R(r) : \Gamma(r) \text{ wval}}{\triangleright_{\Psi} R : \Gamma \text{ regFile}} \text{T_RegBank}
\end{array}$$

The full set of typing rules of *SIFTAL* are organized into the following nine judgements (Figures 3 to 5):

$\triangleright_{\Psi} w : \sigma \text{ wval}$	well-typed word value
$\Gamma \triangleright_{\Psi} v : \sigma \text{ opnd}$	well-typed operand
$\triangleright_{\Psi} h : \sigma \text{ hval}$	well-typed heap value
$\sigma \leq \sigma', \Gamma \leq \Gamma', \Delta \leq \Delta'$	subtyping
$\triangleright_{\Psi} C : \Delta \text{ cstack}$	well-typed continuation stack
$\Xi[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$	well-typed block
$\triangleright H : \Psi \text{ heap}$	well-typed heap
$\triangleright_{\Psi} R : \Gamma \text{ regFile}$	well-typed register bank
$\triangleright \Sigma : \Omega \text{ program}$	well-typed machine

$$\begin{array}{c}
\frac{}{\triangleright_{\Psi} i^l : \text{int}^l \text{ wval}} \text{T_IntLit} \qquad \frac{}{\triangleright_{\Psi} p : \Psi(p) \text{ wval}} \text{T_TplLbl} \quad \frac{L \text{ non-linear}}{\triangleright_{\Psi} L : \Psi(L) \text{ wval}} \text{T_CodeLbl} \\
\\
\frac{\triangleright_{\Psi} w : \sigma \text{ wval}}{\Gamma \triangleright_{\Psi} w : \sigma \text{ opnd}} \text{T_WordOp} \qquad \frac{}{\Gamma \triangleright_{\Psi} r : \Gamma(r) \text{ opnd}} \text{T_RegOp} \\
\\
\frac{\triangleright_{\Psi} w_0 : \sigma_0 \text{ wval} \quad \dots \quad \triangleright_{\Psi} w_{n-1} : \sigma_{n-1} \text{ wval}}{\triangleright_{\Psi} \langle w_0, \dots, w_{n-1} \rangle : \langle \sigma_0, \dots, \sigma_{n-1} \rangle^l \text{ hval}} \text{T_Tpl} \qquad \frac{\Xi[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}}{\triangleright_{\Psi} B : (\langle \mathbf{pc} \rangle \Xi \rightarrow \{\})^l \text{ hval}} \text{T_CBlk} \\
\\
\frac{}{\triangleright_{\Psi} \epsilon : \epsilon \text{ cstack}} \text{T_CSNil} \qquad \frac{\Psi(\underline{L}) \leq \kappa^l \quad \triangleright_{\Psi} C : \Delta \text{ cstack}}{\triangleright_{\Psi} \underline{L} \cdot C : \kappa^l \cdot \Delta \text{ cstack}} \text{T_CSCons}
\end{array}$$

Figure 3. Typing rules for word values, operands, heap values and continuation stacks

$$\begin{array}{c}
\frac{m \geq n}{\{r_1 : \sigma_1, \dots, r_m : \sigma_m\} \leq \{r_1 : \sigma_1, \dots, r_n : \sigma_n\}} \text{ST_RegBank} \quad \frac{\Gamma \leq \Gamma'}{\langle \mathbf{pc} \rangle \Gamma' \mid \Delta \rightarrow \{\} \leq \langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{\}} \text{ST_Cont} \\
\\
\frac{\tau_1 \leq \tau_2}{\tau_1^l \leq \tau_2^l} \text{ST_Latt} \quad \frac{}{\epsilon \leq \epsilon} \text{ST_CSNil} \quad \frac{\Delta \leq \Delta' \quad \kappa^l \leq \kappa'^l}{\kappa^l \cdot \Delta \leq \kappa'^l \cdot \Delta'} \text{ST_CSCons} \quad \frac{}{\tau \leq \tau} \text{ST_Refl} \quad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \text{ST_Trans}
\end{array}$$

Figure 4. Subtyping rules

The subtyping rules together with those for word values, operands, heap values and continuation stacks present no complications.

Consider the typing rules for code blocks. In T_Jmp , the conditions $\Gamma \leq \Gamma'$ and $\Delta \leq \Delta'$ force the current register bank and continuation stack types to be compatible with the ones expected at the destination code block. The condition $\mathbf{pc} \sqcup l \sqsubseteq \mathbf{pc}'$ avoids implicit flows by requiring that the security level of the destination code block, \mathbf{pc}' , be at least that of the result of joining \mathbf{pc} with the security level required to access the destination code block. In T_Jmpcc the type of the continuation stack of the destination code block should coincide with the type of the current continuation stack. This linearity restriction forces the pending continuation *obligations* to be passed on to the destination code block. In T_Push , the condition $\mathbf{pc} \sqsubseteq \mathbf{pc}'$ prevents the security level of the program counter of the continuation \underline{L} (once called) to be dropped below the current \mathbf{pc} . Accessing a code label for which the current \mathbf{pc} has no security clearance is prevented by condition $\mathbf{pc} \sqsubseteq l$. The remaining block (B') is typed under the new continuation stack type resulting from pushing κ^l onto Δ .

Correct termination of computation must yield and empty stack and end with a `halt` instruction. A program $(H, R, B, \mathbf{pc}, C) : \Omega$ is *stuck* if it is not of the form $(H, R, \text{halt}, \mathbf{pc}, \epsilon) : [\Psi, \Gamma, \epsilon]$ and there does not exist a typed program $\Sigma' : \Omega'$ such that $\Sigma : \Omega \mapsto \Sigma' : \Omega'$. Two important results relating the type system and the opera-

tional semantics are *Progress* (well-typed programs do not get stuck) and *Subject Reduction* (the set of well-typed programs is closed with respect to reduction). Full proofs may be found in [10].

Proposition 2.1 (Progress) If $\triangleright \Sigma : \Omega$ program then either there exists $\Sigma' : \Omega'$ such that $\Sigma : \Omega \mapsto \Sigma' : \Omega'$, or $\Sigma : \Omega$ is of the form $(H, R, \text{halt}, \mathbf{pc}, \epsilon) : [\Psi, \Gamma, \epsilon]$.

Proposition 2.2 (Subject Reduction) If $\triangleright \Sigma : \Omega$ program and $\Sigma : \Omega \mapsto \Sigma' : \Omega'$, then $\triangleright \Sigma' : \Omega'$ program.

3. Non-Interference

The non-interference property states that computed low security level values should not be affected by high security ones. As usual, “low security” and “high security” are relative to an arbitrary security level ζ . Thus “low security” means $\sqsubseteq \zeta$ and “high security” means $\not\sqsubseteq \zeta$. In order to prove that a program satisfies non-interference one must show that any two executions of it that are fired from indistinguishable configurations (with regards to a fixed security observation level ζ) must yield, if they both terminate, indistinguishable configurations of the same security observation level. The key to such a proof is a precise rendering of the notion of indistinguishable configurations.

Since our programs consist of a number of elements (heaps, registers, code and continuation stack), all such

$$\begin{array}{c}
\frac{}{\Gamma \mid \epsilon[\mathbf{pc}] \triangleright_{\Psi} \text{halt blk}} \text{T_Halt} \\
\\
\frac{\Gamma \triangleright_{\Psi} v : (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \text{ opnd} \quad \Gamma \leq \Gamma' \quad \Delta \leq \Delta' \quad \mathbf{pc} \sqcup l \sqsubseteq \mathbf{pc}'}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{jmp } v \text{ blk}} \text{T_Jmp} \\
\\
\frac{\kappa^l = (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \quad \Gamma \leq \Gamma' \quad l \sqsubseteq \mathbf{pc}'}{\Gamma \mid \kappa^l \cdot \Delta'[\mathbf{pc}] \triangleright_{\Psi} \text{jmpcc blk}} \text{T_Jmpcc} \\
\\
\frac{\Gamma \triangleright_{\Psi} r_s : \text{int}^{l_1} \text{ opnd} \quad \Gamma \triangleright_{\Psi} v : \text{int}^{l_2} \text{ opnd} \quad l_1 \sqcup l_2 \sqsubseteq \text{label}(\Gamma(r_d)) \quad \mathbf{pc} \sqsubseteq l_1 \sqcup l_2 \quad \Gamma[r_d := \text{int}^{l_1 \sqcup l_2}] \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{aop } r_d, r_s, v; B' \text{ blk}} \text{T_Arith} \\
\\
\frac{\Gamma \triangleright_{\Psi} r : \text{int}^{l_1} \text{ opnd} \quad \Gamma \triangleright_{\Psi} v : (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^{l_2} \text{ opnd} \quad \Gamma \leq \Gamma' \quad \Delta \leq \Delta' \quad \mathbf{pc} \sqcup l_1 \sqcup l_2 \sqsubseteq \mathbf{pc}' \quad \Gamma \mid \Delta[\mathbf{pc} \sqcup l_1] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{bnz } r, v; B' \text{ blk}} \text{T_CondBrnch} \\
\\
\frac{\Gamma \triangleright_{\Psi} v : \tau^l \text{ opnd} \quad \mathbf{pc} \sqsubseteq l \quad l \sqsubseteq \text{label}(\Gamma(r)) \quad \Gamma[r := \tau^l] \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{mov } r, v; B' \text{ blk}} \text{T_Mov} \\
\\
\frac{\Gamma \triangleright_{\Psi} r_s : \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \text{ opnd} \quad \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i) \quad \text{label}(\sigma_i) \sqsubseteq \text{label}(\Gamma(r_d)) \quad \Gamma[r_d := \sigma_i] \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{ld } r_d, r_s[i]; B' \text{ blk}} \text{T_Ld} \\
\\
\frac{\Gamma \triangleright_{\Psi} r_d : \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \text{ opnd} \quad \Gamma \triangleright_{\Psi} r_s : \sigma_i \text{ opnd} \quad \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i) \quad \Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{st } r_d[i], r_s; B' \text{ blk}} \text{T_St} \\
\\
\frac{\Psi(\underline{L}) = (\langle \mathbf{pc}' \rangle \Gamma_1 \mid \Delta_1 \rightarrow \{\})^l \quad \Psi(\underline{L}) \leq \kappa^l \quad \mathbf{pc} \sqsubseteq \mathbf{pc}' \quad \mathbf{pc} \sqsubseteq l \quad \Gamma \mid \kappa^l \cdot \Delta[\mathbf{pc}] \triangleright_{\Psi} B' \text{ blk}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{cpush } \underline{L}; B' \text{ blk}} \text{T_Push}
\end{array}$$

Figure 5. Typing rules for basic blocks

items must be taken into consideration. Indistinguishable heaps for an observer of security level ζ is relatively straightforward: if two heap values are of low security level, then they should be identical. Since the security level of a heap value is given by its type, such a notion is defined for typed heap values. Note that this also requires defining a notion of indistinguishable word values for an observer of security level ζ . We introduce the following ζ -indistinguishability (Fig. 6-7 and Def. 3.1) judgements for:

$\triangleright_{\Psi_1, \Psi_2} w_1 \approx_{\zeta} w_2 : \sigma \text{ wval}$	typed word values
$\triangleright_{\Psi_1, \Psi_2} h_1 \approx_{\zeta} h_2 : \sigma \text{ hval}$	typed heap values
$\triangleright H_1 \approx_{\zeta} H_2 : \Psi_1 \wedge \Psi_2 \text{ heap}$	heaps
$\triangleright_{\Psi_1, \Psi_2} R_1 \approx_{\zeta} R_2 : \Gamma_1 \wedge \Gamma_2 \text{ regFile}$	register banks
$\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackLow}$	low cont. stacks
$\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh}$	high cont. stacks
$\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : \Omega_1 \wedge \Omega_2 \text{ program}$	programs

Two ζ -indistinguishable programs with low \mathbf{pc} are executing the same instruction and hence they have continuation stacks of the same size. (In this case we use the `cstackLow` judgement). Of course, their continuation stacks need not be identical, since it may be the case that a continuation of a high level is pushed by each and that this

continuation is of different (high) security levels. This explains the typing rule `LowHigh`. However, in the case that the \mathbf{pc} level is high both programs may fork on an instruction such as `bnz` and take different execution paths. In this case, each may add different continuation labels to their respective stacks (all of them high), and thus their sizes may differ. Here we use the `cstackHigh` judgement. However, at all times and until they converge at a linear continuation, they shall have a common “substack” of linear continuations. This is reflected by the `HighAxiom` axiom.

We now address ζ -indistinguishability of programs. Both programs are required to be typable, their respective heaps and register banks must be ζ -indistinguishable at the corresponding types. Also, if their \mathbf{pc} level is low, their respective \mathbf{pcs} must coincide, their code blocks must be identical and their continuation stacks must be ζ -indistinguishable. However, if their \mathbf{pc} level is high, then nothing can be said about the code blocks, but we must require their continuation stacks to be ζ -indistinguishable.

Definition 3.1 (ζ -indistinguishability of programs) Let $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$ and $\Omega_i = [\Psi_i, \Gamma_i, \Delta_i]$, $i \in 1..2$. The judgement $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : \Omega_1 \wedge \Omega_2 \text{ program}$ holds iff

$$\begin{array}{c}
\frac{}{\triangleright \epsilon \approx_{\zeta} \epsilon : \epsilon \wedge \epsilon \text{ cstackLow}} \text{LowAxiom} \\
\\
\frac{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackLow} \quad \kappa_1 = \langle \mathbf{pc}_1 \rangle \Gamma' \mid \Delta' \rightarrow \{\} \quad \kappa_2 = \langle \mathbf{pc}_2 \rangle \Gamma'' \mid \Delta'' \rightarrow \{\}}{\triangleright \underline{L}_1 \cdot C_1 \approx_{\zeta} \underline{L}_2 \cdot C_2 : \kappa_1^{l_1} \cdot \Delta_1 \wedge \kappa_2^{l_2} \cdot \Delta_2 \text{ cstackLow}} \text{LowHigh} \\
\\
\frac{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh} \quad \kappa = \langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{\} \quad \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\triangleright \underline{L} \cdot C_1 \approx_{\zeta} C_2 : \kappa^l \cdot \Delta_1 \wedge \Delta_2 \text{ cstackHigh}} \text{HighLeft} \\
\\
\frac{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackLow} \quad \kappa = \langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{\} \quad \mathbf{pc} \sqcup l \sqsubseteq \zeta}{\triangleright \underline{L} \cdot C_1 \approx_{\zeta} \underline{L} \cdot C_2 : \kappa^l \cdot \Delta_1 \wedge \kappa^l \cdot \Delta_2 \text{ cstackLow}} \text{LowLow} \\
\\
\frac{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackLow}}{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh}} \text{HighAxiom} \\
\\
\frac{\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh} \quad \kappa = \langle \mathbf{pc} \rangle \Gamma \mid \Delta \rightarrow \{\} \quad \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\triangleright C_1 \approx_{\zeta} \underline{L} \cdot C_2 : \Delta_1 \wedge \kappa^l \cdot \Delta_2 \text{ cstackHigh}} \text{HighRight}
\end{array}$$

Figure 6. ζ -indistinguishability of stacks

$$\begin{array}{c}
\frac{\triangleright_{\Psi_1} w_1 : \tau^l \text{ wval} \quad \triangleright_{\Psi_2} w_2 : \tau^l \text{ wval} \quad l_1 \sqsubseteq \zeta \Rightarrow w_1 = w_2}{\triangleright_{\Psi_1, \Psi_2} w_1 \approx_{\zeta} w_2 : \tau^l \text{ wval}} \text{Eq_wval} \quad \frac{\triangleright_{\Psi_1} \langle \bar{w} \rangle : \langle \bar{\rho} \rangle^l \text{ hval} \quad \triangleright_{\Psi_2} \langle \bar{w}' \rangle : \langle \bar{\rho} \rangle^l \text{ hval} \quad l \sqsubseteq \zeta \Rightarrow \triangleright_{\Psi_1, \Psi_2} w_i \approx_{\zeta} w'_i : \sigma_i \text{ wval}, 1 \leq i \leq n}{\triangleright_{\Psi_1, \Psi_2} \langle \bar{w} \rangle \approx_{\zeta} \langle \bar{w}' \rangle : \langle \bar{\rho} \rangle^l \text{ hval}} \text{Eq_hval_tuple} \\
\\
\frac{\Xi[\mathbf{pc}] \triangleright_{\Psi_1} B_1 \text{ blk} \quad \Xi[\mathbf{pc}] \triangleright_{\Psi_2} B_2 \text{ blk} \quad l \sqsubseteq \zeta \Rightarrow B_1 = B_2}{\triangleright_{\Psi_1, \Psi_2} B_1 \approx_{\zeta} B_2 : (\langle \mathbf{pc} \rangle \Xi \rightarrow \{\})^l \text{ hval}} \text{Eq_hval_blk}
\end{array}$$

$\triangleright H_1 \approx_{\zeta} H_2 : \Psi_1 \wedge \Psi_2 \text{ heap}$ iff for all $\ell \in \text{Dom}_{\cup}(\Psi_1, \Psi_2)$ the following holds :

$$\text{label}(\Psi_1(\ell)) \sqsubseteq \zeta \text{ or } \text{label}(\Psi_2(\ell)) \sqsubseteq \zeta, \text{ implies } \begin{cases} \ell \in \text{Dom}_{\cap}(H_1, H_2, \Psi_1, \Psi_2), \text{ and} \\ \Psi_1(\ell) = \Psi_2(\ell), \\ \triangleright_{\Psi_1, \Psi_2} H_1(\ell) \approx_{\zeta} H_2(\ell) : \Psi_1(\ell) \text{ hval} \end{cases}$$

$\triangleright_{\Psi_1, \Psi_2} R_1 \approx_{\zeta} R_2 : \Gamma_1 \wedge \Gamma_2 \text{ regFile}$ iff for all $r \in \text{Dom}_{\cup}(\Gamma_1, \Gamma_2)$ the following holds :

$$\text{label}(\Gamma_1(r)) \sqsubseteq \zeta \text{ or } \text{label}(\Gamma_2(r)) \sqsubseteq \zeta, \text{ implies } \begin{cases} r \in \text{Dom}_{\cap}(R_1, R_2, \Gamma_1, \Gamma_2), \text{ and} \\ \Gamma_1(r) = \Gamma_2(r), \text{ and} \\ \triangleright_{\Psi_1, \Psi_2} R_1(r) \approx_{\zeta} R_2(r) : \Gamma_1(r) \text{ wval} \end{cases}$$

Note: $\text{Dom}_{\oplus}(A_1, \dots, A_n)$ abbreviates $\text{Dom}(A_1) \oplus \dots \oplus \text{Dom}(A_n)$

Figure 7. ζ -indistinguishability

1. $\triangleright \Sigma_1 : \Omega_1 \text{ program}$ and $\triangleright \Sigma_2 : \Omega_2 \text{ program}$
2. $\triangleright H_1 \approx_{\zeta} H_2 : \Psi_1 \wedge \Psi_2 \text{ heap}$
3. $\triangleright_{\Psi_1, \Psi_2} R_1 \approx_{\zeta} R_2 : \Gamma_1 \wedge \Gamma_2 \text{ regFile}$
4. (a) either, $\mathbf{pc}_1 = \mathbf{pc}_2 \sqsubseteq \zeta$ and $B_1 = B_2$ and $\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackLow}$,
(b) or, $\mathbf{pc}_1 \not\sqsubseteq \zeta$ and $\mathbf{pc}_2 \not\sqsubseteq \zeta$ and $\triangleright C_1 \approx_{\zeta} C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh}$

Non-interference states that heap and register values that are of a high security level do not affect computed low security ones.

Theorem 3.1 (Non-Interference) Given typed machine configurations $\Sigma_1 = (H_1, R_1, B, \perp, \epsilon) : [\Psi, \Gamma, \Delta]$ and $\Sigma_2 = (H_2, R_2, B, \perp, \epsilon) : [\Psi, \Gamma, \Delta]$. If the following conditions hold

- $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : [\Psi, \Gamma, \Delta] \wedge [\Psi, \Gamma, \Delta] \text{ program}$
- $\Sigma'_1 = (H'_1, R'_1, \text{halt}, \zeta, \epsilon)$ and $\Sigma_1 : [\Psi, \Gamma, \Delta] \xrightarrow{*} \Sigma'_1 : [\Psi'_1, \Gamma'_1, \epsilon]$
- $\Sigma'_2 = (H'_2, R'_2, \text{halt}, \zeta, \epsilon)$ and $\Sigma_2 : [\Psi, \Gamma, \Delta] \xrightarrow{*} \Sigma'_2 : [\Psi'_2, \Gamma'_2, \epsilon]$

Then $\triangleright \Sigma'_1 \approx_{\zeta} \Sigma'_2 : [\Psi'_1, \Gamma'_1, \epsilon] \wedge [\Psi'_2, \Gamma'_2, \epsilon] \text{ program}$.

Note that we start off from two heaps H_1 and H_2 that may differ in the contents of the heap locations of high security types (and similarly for the register banks). More precisely, for all those ℓ such that $\text{label}(\Psi(\ell)) \not\sqsubseteq \zeta$, the heap values $H_1(\ell)$ and $H_2(\ell)$ may differ. However, those considered of low security type (i.e. those ℓ such that $\text{label}(\Psi(\ell)) \sqsubseteq \zeta$) must be identical.

We assume that computation does not diverge and arrive at programs whose **pc** security level is ζ . All those computed heap and register values of low security type (i.e. $\sqsubseteq \zeta$) must then be identical.

Having defined the notion of ζ -indistinguishable programs, the technical challenge that lies in the proof of Theorem 3.1 is that this equivalence holds after each transition step. When the **pc** is of low level, the programs shall be executing the same instructions (with possibly different heap, register bank and continuation stacks). They may be seen to be *synchronized* and each reduction step made by one shall by emulated with a reduction of the exact same instruction by the other. The resulting programs must of course be ζ -indistinguishable once again.

However, because of a **brnz**, it is possible that the execution sequences of the programs fork on a high value. As a consequence, both their **pcs** shall become high and once again we must provide proof that there are some programs to which they reduce and at which point they are once again ζ -indistinguishable.

Thus the proof is developed in two stages. First it is proved that two ζ -indistinguishable programs that have a low (and identical) **pc** level can reduce in a *lock step fashion* in a manner invariant to the ζ -indistinguishability property.

Lemma 3.2 (Low-pc step) Let $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$, $i \in 1..2$. If

1. $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2]$ program, for some heap, register and stack types Ψ_i, Γ_i and Δ_i , respectively.
2. $\mathbf{pc}_1 \sqsubseteq \zeta$ and $\mathbf{pc}_2 \sqsubseteq \zeta$
3. $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$, for some heap, register and stack types Ψ'_1, Γ'_1 and Δ'_1 , respectively.

Then there exists a program $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ such that

- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \mapsto \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ and
- $\triangleright \Sigma'_1 \approx_{\zeta} \Sigma'_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi'_2, \Gamma'_2, \Delta'_2]$ program, for some heap, register and stack types Ψ'_2, Γ'_2 and Δ'_2 , respectively.

In the case of ζ -indistinguishable programs of high **pc** each program may be allowed to execute different instructions. In particular, new continuations may be added to the continuation stack and then popped by one program, while the other leaves its continuation stack as is. However, notice that, as already discussed, both programs shall share a common “substack”, namely Δ' in the following definition.

Definition 3.2 A continuation stack C' of type Δ' is said to be ζ -topped in C of type Δ , if there exist linear labels $\underline{L}_1, \dots, \underline{L}_n$ (possibly none), and continuation types $\kappa_1^{l_1}, \dots, \kappa_n^{l_n}$ (possibly none) such that:

- $C = \underline{L}_1 \cdot \dots \cdot \underline{L}_n \cdot C'$
- $\Delta = \kappa_1^{l_1} \cdot \dots \cdot \kappa_n^{l_n} \cdot \Delta'$ and
- $\kappa_i = \langle \mathbf{pc}'_i \rangle \Gamma'_i \mid \Delta'_i \rightarrow \{\}$ implies $\mathbf{pc}'_i \sqcup l_i \not\sqsubseteq \zeta$, for all $1 \leq i \leq n$.

Also in the case where the **pc** level is high, the type rules guarantee that all changes made to the heap and register bank must be of a high security level. Thus, in the case where the **pc** security level is high, the following invariant is seen to hold.

Lemma 3.3 (High-step invariant) Suppose

1. $\Sigma_1 : \Omega_1 \xrightarrow{*} \Sigma_k : \Omega_k$, for $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$ and $\Omega_i = [\Psi_i, \Gamma_i, \Delta_i]$, for all $1 \leq i \leq k$.
2. $\triangleright \Sigma_1 : \Omega_1$ program
3. $\mathbf{pc}_1 \not\sqsubseteq \zeta$
4. C_k of type Δ_k is ζ -topped in C_i of type Δ_i , for each $1 \leq i \leq k$.

Then

- $\triangleright H_1 \approx_{\zeta} H_k : \Psi_1 \wedge \Psi_k$ heap
- $\triangleright_{\Psi_1, \Psi_k} R_1 \approx_{\zeta} R_k : \Gamma_1 \wedge \Gamma_k$ regFile
- $\triangleright C_1 \approx_{\zeta} C_k : \Delta_1 \wedge \Delta_k$ cstackHigh
- $\mathbf{pc}_i \not\sqsubseteq \zeta$, for all $1 \leq i \leq k$.

This invariant is the main ingredient required for the proof of the High-pc Step Lemma, the second stage of the proof of Theorem 3.1.

Lemma 3.4 (High-pc Step) Let $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$, $i \in 1..2$. If

- $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2]$ program, for some heap, register and stack types Ψ_i, Γ_i and Δ_i , respectively.
- $\mathbf{pc}_1 \not\sqsubseteq \zeta$ and $\mathbf{pc}_2 \not\sqsubseteq \zeta$
- $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$, for some heap, register and stack types Ψ'_1, Γ'_1 and Δ'_1 , respectively.

Then either the program $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ diverges or there exists a typed program $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ such that

- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \xrightarrow{*} \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ and
- $\triangleright \Sigma'_1 \approx_{\zeta} \Sigma'_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi'_2, \Gamma'_2, \Delta'_2]$ program, for some heap, register and stack types Ψ'_2, Γ'_2 and Δ'_2 , respectively.

Finally, the proof of the Non-interference Theorem (Thm. 3.1) is attained by weaving low and high **pc** steps and using Lemmas 3.2 and 3.4.

4. Conclusions and Future Work

We defined *SIFTAL*, a typed assembly language for secure information flow analysis. Besides the standard data-structures of an assembly language such as heap and register bank, *SIFTAL* has a stack of linear continuations, which together with the special instructions `cpush` and `jmpcc` allow the compilation of high-level control structures – such as *if-then-else* –, simulating at the assembly level the block structure of the source language.

We define a notion of ζ -indistinguishability of machine configurations where high-level data is indistinguishable: two machine configurations are ζ -indistinguishable if they coincide at observable (low-level) values. The type system guarantees that typable machine configurations satisfy the non-interference property (Theorem 3.1): if two typable machine configuration are ζ -indistinguishable, then the resulting machine configurations after execution are also ζ -indistinguishable.

This is the first typed assembly language with linear continuations and security types for information flow analysis, and our proof of non-interference is the first such proof for a MIPS-style typed assembly language.

Among the open problems we are studying is the combination of safety and information flow analysis in the same type system, and the extension of *SIFTAL* with an execution stack.

5. Acknowledgments

We are grateful to Pablo Garralda and Healfdene Goguen for enlightening discussions and comments on previous drafts.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [3] D. Aspinall and A. B. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 2004. Special Issue on Proof-Carrying Code. To appear.
- [4] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [5] G. Barthe and T. Rezk. Information flow for a sequential Java Virtual Machine. Unpublished, 2003.
- [6] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report Technical Report MTR 2547 v2, MITRE, November 1973.
- [7] C. Bernardeschi and N. D. Francesco. Combining abstract interpretation and model checking for analysing security properties of Java bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [8] K. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.
- [9] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking secure interactions of smart card applets: Extended version. *Journal of Computer Security*, 10(4):369–398, 2002.
- [10] E. Bonelli, A. B. Compagnoni, and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. <http://guinness.cs.stevens-tech.edu/~ebonelli/siftalLong.ps.gz>, 2004.
- [11] J. Cheney and G. Morrisett. A linearly typed assembly language.
- [12] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pages 25–35, May 1999.
- [13] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, May 1976.
- [14] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
- [15] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *6th ACM Symp. Operating System Principles*, pages 57–65, November 1977.
- [16] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1982.
- [17] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86. IEEE Press, 1984.
- [18] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, Mar. 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.
- [19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
- [20] G. Necula. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1997.
- [21] P. G. Neumann, R. J. Feiertag, K. N. Levitt, and L. Robinson. Software development and proofs of multi-level security. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 421–428. IEEE Computer Society, October 1976.
- [22] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

- [23] F. Smith, D. Walker, and G. Morrisett. Alias types. In G. Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, Apr. 2000.
- [24] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
- [25] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Third International Workshop on Types in Compilation*, Montreal, Canada, Sept. 2000.
- [26] H. Xi and R. Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology, July 1999.
- [27] S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3), 2002.

A. Auxiliary Lemmas

In this section we provide, without proof (see [10] for full proofs), a series of lemmas used in the proofs of the main results.

Lemma A.1 (Relating Ψ and Γ) If $\triangleright_{\Psi} R : \Gamma \text{ regFile}$ and $r \in \text{Dom}(\Gamma)$ and $R(r) = \ell$, then $\Psi(\ell) = \Gamma(r)$.

Lemma A.2 (\hat{R} Typing) If $\triangleright_{\Psi} R : \Gamma \text{ regFile}$ and $\Gamma \triangleright_{\Psi} v : \sigma \text{ opnd}$, then $\triangleright_{\Psi} \hat{R}(v) : \sigma \text{ wval}$.

Lemma A.3 (Code block weakening I) If $\Gamma_2 \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$ and $\Gamma_1 \leq \Gamma_2$, then $\Gamma_1 \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$.

Lemma A.4 (Code block weakening II) If $\Gamma \mid \Delta_2[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$ and $\Delta_1 \leq \Delta_2$, then $\Gamma \mid \Delta_1[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$.

Lemma A.5 (Canonical Word Forms)

If $\triangleright H : \Psi \text{ heap}$ and $\triangleright_{\Psi} w : \sigma \text{ wval}$ then:

1. if $\sigma = \text{int}^l$ then $w = i^l$
2. if $\sigma = \langle \sigma_0, \dots, \sigma_{n-1} \rangle^l$ then
 - (a) $w = p$
 - (b) $H(p) = \langle w_0, \dots, w_{n-1} \rangle$
 - (c) $\triangleright_{\Psi} w_i : \sigma_i \text{ wval}$, for all $0 \leq i < n$
3. if $\sigma = (\langle \mathbf{pc} \rangle \Xi \rightarrow \{\})^l$ then
 - (a) $w = L$
 - (b) $H(L) = B$
 - (c) $\Xi[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$

Lemma A.6 (Canonical Forms)

If $\triangleright H : \Psi \text{ heap}$, $\triangleright_{\Psi} R : \Gamma \text{ regFile}$ and $\Gamma \triangleright_{\Psi} v : \sigma \text{ opnd}$, then:

1. if $\sigma = \text{int}^l$ then $\hat{R}(v) = i^l$
2. if $\sigma = \langle \sigma_0, \dots, \sigma_{n-1} \rangle^l$ then
 - (a) $\hat{R}(v) = p$
 - (b) $H(p) = \langle w_0, \dots, w_{n-1} \rangle$
 - (c) $\triangleright_{\Psi} w_i : \sigma_i \text{ wval}$, for all $0 \leq i < n$
3. if $\sigma = (\langle \mathbf{pc} \rangle \Xi \rightarrow \{\})^l$ then
 - (a) $\hat{R}(v) = L$
 - (b) $H(L) = B$
 - (c) $\Xi[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk}$

Lemma A.7 (Auxiliar) $\Gamma \triangleright_{\Psi} v : \sigma \text{ opnd}$ implies $(\Psi \cup \Gamma)(v) = \sigma$.

B. Progress and Subject Reduction Results

Proposition B.1 (Progress) If $\triangleright \Sigma : \Omega$ program then either there exists $\Sigma' : \Omega'$ such that $\Sigma : \Omega \mapsto \Sigma' : \Omega'$, or $\Sigma : \Omega$ is of the form $(H, R, \text{halt}, \mathbf{pc}, \epsilon) : [\Psi, \Gamma, \epsilon]$.

Proof: The proof is by cases on the first instruction of the current code block B of the program $\Sigma : \Omega = (H, R, B, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta]$. As an example, here we include only the case for the `st` instruction, where the typing derivation of $\triangleright \Sigma : \Omega$ program has the form:

$$\frac{\begin{array}{c} \pi \\ \triangleright H : \Psi \text{ heap} \quad \triangleright_{\Psi} R : \Gamma \text{ regFile} \quad \Gamma \mid \Delta \triangleright_{\Psi} \text{st } r_d[i], r_s; B' \text{ blk} \quad \triangleright_{\Psi} C : \Delta \text{ cstack} \end{array}}{\triangleright (H, R, \text{st } r_d[i], r_s; B', \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \text{ program}}$$

where π is the typing derivation:

$$\frac{\begin{array}{c} \Gamma \triangleright_{\Psi} r_d : \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \text{ opnd} \quad \Gamma \triangleright_{\Psi} r_s : \sigma_i \text{ opnd} \\ \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i) \quad \Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} B \text{ blk} \end{array}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{st } r_d[i], r_s; B \text{ blk}} \text{ T_St}$$

Since $\Gamma \triangleright_{\Psi} r_d : \langle \sigma_0, \dots, \sigma_{n-1} \rangle^l \text{ opnd}$, by the Canonical Forms Lemma (Lemma A.6), $\hat{R}(r_d) = p$ (and hence $R(r_d) = p$) for some tuple pointer p , $H(p) = \langle w_0, \dots, w_{n-1} \rangle$ and $\triangleright_{\Psi} w_j : \tau_j \text{ wval}$.

Since $\Gamma \triangleright_{\Psi} r_s : \sigma_i \text{ opnd}$, by \hat{R} Typing Lemma (Lemma A.2) we deduce $\triangleright_{\Psi} \hat{R}(r_s) : \tau_i \text{ wval}$ and, therefore, that $R(r_s)$ is defined. Then we set $\Sigma' : \Omega' = (H[p := \langle w_0, \dots, \hat{R}(r_s), \dots, w_{n-1} \rangle], R, B', \mathbf{pc}, C) : [\Psi, \Gamma, \Delta]$ and

$$\frac{\begin{array}{c} \hat{R}(r_d) = p \quad H(p) = \langle w_0, \dots, w_i, \dots, w_{n-1} \rangle \\ \Gamma(r_d) = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \quad \Gamma(r_s) = \sigma_i \\ \mathbf{pc} \sqcup l \sqsubseteq \text{label}(\sigma_i) \end{array}}{(H, R, \text{st } r_d[i], r_s; B', \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H[p := \langle w_0, \dots, \hat{R}(r_s), \dots, w_{n-1} \rangle], R, B', \mathbf{pc}, C) : [\Psi, \Gamma, \Delta]} \text{ OS_Store}$$

Note that the σ_i in T_St and in OS_Store are one and the same by the Auxiliar Lemma (Lemma A.7). Likewise for $\langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l$ in T_St and in OS_Store.

□

Proposition B.2 (Subject Reduction) If $\triangleright \Sigma : \Omega$ program and $\Sigma : \Omega \mapsto \Sigma' : \Omega'$, then $\triangleright \Sigma' : \Omega'$ program.

Proof: Suppose $\Sigma = (H, R, B, \mathbf{pc}, C)$. The proof is by cases on the reduction step $\Sigma : \Omega \mapsto \Sigma' : \Omega'$. As an example, here we include only the case for the `jmp` instruction, where the typing derivation of Σ has the form:

$$\frac{\begin{array}{c} \pi \\ \triangleright H : \Psi \text{ heap} \quad \triangleright_{\Psi} R : \Gamma \text{ regFile} \quad \Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{jmp } v \text{ blk} \quad \triangleright_{\Psi} C : \Delta \text{ cstack} \end{array}}{\triangleright (H, R, \text{jmp } v, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta]} \text{ T_Program}$$

where the typing derivation π is:

$$\frac{\begin{array}{c} \Gamma \triangleright_{\Psi} v : (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \text{ opnd} \quad \Gamma \leq \Gamma' \\ \Delta \leq \Delta' \quad \mathbf{pc} \sqcup l \sqsubseteq \mathbf{pc}' \end{array}}{\Gamma \mid \Delta[\mathbf{pc}] \triangleright_{\Psi} \text{jmp } v \text{ blk}} \text{ T_Jmp}$$

The reduction step $\Sigma : \Omega \mapsto \Sigma' : \Omega'$ is of the form:

$$\frac{\hat{R}(v) = L \quad H(L) = B_1 \quad (\Psi \cup \Gamma)(v) = (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l}{(H, R, \text{jmp } v, \mathbf{pc}, C) : [\Psi, \Gamma, \Delta] \mapsto (H, R, B_1, \mathbf{pc} \sqcup \mathbf{pc}'_1 \sqcup l, C) : [\Psi, \Gamma, \Delta]} \text{ OS_Jmp}$$

Notice that:

- $\triangleright H : \Psi \text{ heap}$ holds by hypothesis.

- $\triangleright_{\Psi} R : \Gamma \text{ regFile}$ holds by hypothesis.
- $\triangleright_{\Psi} C : \Delta \text{ cstack}$ holds by hypothesis.

So, we are left to prove that

$$\Gamma \mid \Delta[\mathbf{pc} \sqcup \mathbf{pc}'_1 \sqcup l_1] \triangleright_{\Psi} B_1 \text{ blk}$$

Since $\Gamma \triangleright_{\Psi} v : (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \text{ opnd}$, by \hat{R} Typing Lemma (Lemma A.2) we deduce $\triangleright_{\Psi} \hat{R}(v) : (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \text{ wval}$ and by Canonical Word Forms Lemma (Lemma A.5) we deduce

- $\hat{R}(v) = L$
- $H(L) = B_1$
- $\Gamma' \mid \Delta'[\mathbf{pc}'] \triangleright_{\Psi} B_1 \text{ blk}$

Moreover, from $\mathbf{pc} \sqcup l \sqsubseteq \mathbf{pc}'$ we know that $\mathbf{pc}' = \mathbf{pc} \sqcup \mathbf{pc}' \sqcup l$. So the last item may be stated equivalently as

$$\Gamma' \mid \Delta'[\mathbf{pc} \sqcup \mathbf{pc}' \sqcup l] \triangleright_{\Psi} B_1 \text{ blk} \quad (1)$$

Recall from above that we have to prove that $\Gamma \mid \Delta[\mathbf{pc} \sqcup \mathbf{pc}' \sqcup l] \triangleright_{\Psi} B_1 \text{ blk}$. This follows as a result of the following procedure

1. First obtain $\Gamma \mid \Delta'[\mathbf{pc} \sqcup \mathbf{pc}' \sqcup l] \triangleright_{\Psi} B_1 \text{ blk}$ from applying the Code Block Weakening Lemma I (Lemma A.3) to (1) and using $\Gamma \leq \Gamma'$.
2. Finally, obtain $\Gamma \mid \Delta[\mathbf{pc} \sqcup \mathbf{pc}' \sqcup l] \triangleright_{\Psi} B_1 \text{ blk}$ from applying the Code Block Weakening Lemma II (Lemma A.4) to (1) and using $\Delta \leq \Delta'$.

□

C. Proof of Non-Interference

Lemma C.1 (Low-pc step) Let $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$, $i \in 1..2$. If the following conditions hold

1. $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2] \text{ program}$, for some heap, register and stack types Ψ_i, Γ_i and Δ_i , respectively.
2. $\mathbf{pc}_1 \sqsubseteq \zeta$ and $\mathbf{pc}_2 \sqsubseteq \zeta$
3. $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$, for some heap, register and stack types Ψ'_1, Γ'_1 and Δ'_1 , respectively.

Then there exists a program $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ such that

- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \mapsto \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ and
- $\triangleright \Sigma'_1 \approx_{\zeta} \Sigma'_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi'_2, \Gamma'_2, \Delta'_2] \text{ program}$, for some heap, register and stack types Ψ'_2, Γ'_2 and Δ'_2 , respectively.

Proof: The proof goes by case analysis on the instruction executed in the step $\Sigma_1 : \Omega_1 \mapsto \Sigma'_1 : \Omega'_1$.

Note that since $\triangleright \Sigma_1 \approx_{\zeta} \Sigma_2 : \Omega_1 \wedge \Omega_2 \text{ program}$, $\mathbf{pc}_1 \sqsubseteq \zeta$ and $\mathbf{pc}_2 \sqsubseteq \zeta$, we know that $\mathbf{pc}_1 = \mathbf{pc}_2$. Thus, for simplicity, we shall use \mathbf{pc} for both.

As an example, here we include only the interesting case for the `st` instruction, where $B_1 = \text{st } r_d[i], r_s; B$ and the step $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$ takes the form:

$$\begin{array}{ll} \hat{R}_1(r_d) = p_1 & H_1(p_1) = \langle w_0, \dots, w_i, \dots, w_{n-1} \rangle \\ \Gamma_1(r_d) = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^{l_1} & \Gamma_1(r_s) = \sigma_i \\ \mathbf{pc} \sqcup l_1 \sqsubseteq \text{label}(\sigma_i) & \end{array}$$

OS_Store

$$(H_1, R_1, \text{st } r_d[i], r_s; B, \mathbf{pc}, C_1) : [\Psi_1, \Gamma_1, \Delta_1] \mapsto (H_1[p_1 := \langle w_0, \dots, \hat{R}_1(r_s), \dots, w_{n-1} \rangle], R_1, B, \mathbf{pc}, C_1) : [\Psi_1, \Gamma_1, \Delta_1]$$

By hypothesis (1) and (2) we know that $B_1 = B_2$ and then $\Sigma_2 = (H_2, R_2, \text{st } r_d[i], r_s; B, \mathbf{pc}, C_2)$.

By hypothesis (1) we know that $\triangleright \Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ program, then, by Progress (Proposition 2.1), we know that there exists a program Σ'_2 such that $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \mapsto \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$. This machine configuration is $\Sigma'_2 = (H'_2, R_2, B, \mathbf{pc}, C_2) : [\Psi'_2, \Gamma'_2, \Delta'_2]$ where $H'_2 = H_2[p_2 := \langle w'_0, \dots, \hat{R}_2(r_s), \dots, w'_{m-1} \rangle]$, $\Psi'_2 = \Psi_2$, $\Gamma'_2 = \Gamma_2$, $\Delta'_2 = \Delta_2$ and

$$\frac{\begin{array}{l} \hat{R}_2(r_d) = p_2 \\ \Gamma_2(r_d) = \langle \sigma'_0, \dots, \sigma'_i, \dots, \sigma'_{m-1} \rangle^{l_2} \\ \mathbf{pc} \sqcup l_2 \sqsubseteq \mathbf{label}(\sigma'_i) \end{array} \quad \begin{array}{l} H_2(p_2) = \langle w'_0, \dots, w'_i, \dots, w'_{m-1} \rangle \\ \Gamma_2(r_s) = \sigma'_i \end{array}}{(H_2, R_2, \mathbf{st} \ r_d[i], r_s; B, \mathbf{pc}, C_2) : [\Psi_2, \Gamma_2, \Delta_2] \mapsto (H_2[p_2 := \langle w'_0, \dots, \hat{R}_2(r_s), \dots, w'_{m-1} \rangle], R_2, B, \mathbf{pc}, C_2) : [\Psi_2, \Gamma_2, \Delta_2]} \text{OS_Store}$$

Note that:

- $\triangleright \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$ program and $\triangleright \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ program hold by Subject Reduction (Proposition 2.2).
- Given $\mathbf{pc} \sqsubseteq \zeta$ and given in both Σ'_1 and Σ'_2 the code block is B , it is enough to show that since $\Delta'_1 = \Delta_1$ and $\Delta'_2 = \Delta_2$, by hypothesis (1) $\triangleright C_1 \approx_\zeta C_2 : \Delta'_1 \wedge \Delta'_2$ cstackLow
- $\triangleright_{\Psi'_1, \Psi'_2} R'_1 \approx_\zeta R'_2 : \Gamma'_1 \wedge \Gamma'_2$ regFile holds by hypothesis since $\Psi'_i = \Psi_i$, $R'_i = R_i$ and $\Gamma'_i = \Gamma_i$, for $i \in 1..2$.

Thus we are left to verify that $\triangleright H'_1 \approx_\zeta H'_2 : \Psi'_1 \wedge \Psi'_2$ heap where $\Psi'_1 = \Psi_1$ and $\Psi'_2 = \Psi_2$. To do so we have two cases to consider:

- Suppose that $\mathbf{label}(\Psi'_1(p_1)) \sqsubseteq \zeta$. Note that $\Psi'_1(p_1) = \Psi_1(p_1)$. Since $\hat{R}_1(r_d) = p_1$, by Relating Ψ and Γ Lemma (Lemma A.1) we know that $\Psi_1(p_1) = \Gamma_1(r_d)$ and $\mathbf{label}(\Psi_1(p_1)) = \mathbf{label}(\Gamma_1(r_d))$. Thus, since $\mathbf{label}(\Gamma_1(r_d)) \sqsubseteq \zeta$ and by ζ -equivalence of R_1 and R_2 , we may deduce that $\mathbf{label}(\Gamma_2(r_d)) \sqsubseteq \zeta$, $R_1(r_d) = p_1 = p_2 = R_2(r_d)$, and $\mathbf{label}(\Psi_2(p_2)) \sqsubseteq \zeta$.

So, we need to prove that $\triangleright H'_1(p_1) \approx_\zeta H'_2(p_2) : \Psi'_1$ hval. To do that, we only have to analyze the i^{th} element in the tuple, and we have two cases to consider:

- If $\mathbf{label}(\Gamma_1(r_s)) = \mathbf{label}(\sigma_i) \sqsubseteq \zeta$, then, by $\triangleright_{\Psi_1, \Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regFile, we know that $\mathbf{label}(\Gamma_2(r_s)) = \mathbf{label}(\sigma'_i) \sqsubseteq \zeta$ and $\triangleright_{\Psi_1, \Psi_2} R_1(r_s) \approx_\zeta R_2(r_s) : \Gamma_1(r_s)$ wval. So, we deduce $\hat{R}_1(r_s) = \hat{R}_2(r_s)$ and we conclude.
- If $\mathbf{label}(\Gamma_1(r_s)) = \mathbf{label}(\sigma_i) \not\sqsubseteq \zeta$, then, by $\triangleright_{\Psi_1, \Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regFile, we know that $\mathbf{label}(\Gamma_2(r_s)) = \mathbf{label}(\sigma'_i) \not\sqsubseteq \zeta$. But, since $\sigma_i = \sigma'_i$, we know that $\triangleright_{\Psi_1, \Psi_2} R_1(r_s) \approx_\zeta R_2(r_s) : \sigma_i$ wval, $\mathbf{label}(\sigma_i) \not\sqsubseteq \zeta$ and $\mathbf{label}(\sigma'_i) \not\sqsubseteq \zeta$. Then we deduce that $\triangleright_{\Psi'_1, \Psi'_2} H'_1(p_1) \approx_\zeta H'_2(p_2) : \Psi'_1(p_1)$ hval and conclude.

- Suppose $\mathbf{label}(\Psi'_1(p_1)) \not\sqsubseteq \zeta$. Note that, by rule OS_Store, $\mathbf{label}(\Psi'_1(p_1)) = l_1 = \mathbf{label}(\Gamma_1(r_d))$. Then, by $\triangleright_{\Psi_1, \Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regFile, we know that $\mathbf{label}(\Psi'_2(p_2)) = l_2 = \mathbf{label}(\Gamma_2(r_d)) \not\sqsubseteq \zeta$ and we conclude.

□

Lemma C.2 (High-step invariant) Suppose

1. $\Sigma_1 : \Omega_1 \mapsto^* \Sigma_k : \Omega_k$, for $\Sigma_i = (H_i, R_i, B_i, \mathbf{pc}_i, C_i)$ and $\Omega_i = [\Psi_i, \Gamma_i, \Delta_i]$, for all $1 \leq i \leq k$.
2. $\triangleright \Sigma_1 : \Omega_1$ program
3. $\mathbf{pc}_1 \not\sqsubseteq \zeta$
4. C_k of type Δ_k is ζ -topped in C_i of type Δ_i , for each $1 \leq i \leq k$.

Then

- $\triangleright H_1 \approx_\zeta H_k : \Psi_1 \wedge \Psi_k$ heap
- $\triangleright_{\Psi_1, \Psi_k} R_1 \approx_\zeta R_k : \Gamma_1 \wedge \Gamma_k$ regFile
- $\triangleright C_1 \approx_\zeta C_k : \Delta_1 \wedge \Delta_k$ cstackHigh
- $\mathbf{pc}_i \not\sqsubseteq \zeta$, for all $1 \leq i \leq k$.

Proof: By induction on the number of reduction steps m in $\Sigma_1 : \Omega_1 \xrightarrow{*} \Sigma_k : \Omega_k$.

- $m = 0$. In this case $\triangleright H_1 \approx_\zeta H_k : \Psi_1 \wedge \Psi_k$ heap follows from reflexivity of ζ -equivalence on heaps. Likewise for register files and continuation stacks. Also, $\mathbf{pc}_1 = \mathbf{pc}_k$ and $\mathbf{pc}_k \not\sqsubseteq \zeta$ follows from the hypothesis that $\mathbf{pc}_1 \not\sqsubseteq \zeta$.
- $m > 0$. We reason by cases on the reduction rule $\Sigma_1 : \Omega_1 \mapsto \Sigma_2 : \Omega_2$.

As an example, here we include only the interesting case for the `st` instruction, where the reduction step is:

$$\frac{\begin{array}{l} \hat{R}_1(r_d) = p \quad H_1(p) = \langle w_0, \dots, w_i, \dots, w_{n-1} \rangle \\ \Gamma_1(r_d) = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_{n-1} \rangle^l \quad \Gamma_1(r_s) = \sigma_i \\ \mathbf{pc}_1 \sqcup l \sqsubseteq \text{label}(\sigma_i) \end{array}}{(H_1, R_1, \text{st } r_d[i], r_s; B, \mathbf{pc}_1, C_1) : [\Psi_1, \Gamma_1, \Delta_1] \mapsto (H_1[p := h], R_1, B, \mathbf{pc}_1, C_1) : [\Psi_1, \Gamma_1, \Delta_1]} \text{OS_Store}$$

where $h = \langle w_0, \dots, \hat{R}_1(r_s), \dots, w_{n-1} \rangle$. Since $\mathbf{pc}_2 = \mathbf{pc}_1 \not\sqsubseteq \zeta$, we may apply the induction hypothesis and deduce that:

- $\triangleright H_2 \approx_\zeta H_k : \Psi_2 \wedge \Psi_k$ heap
- $\triangleright_{\Psi_2, \Psi_k} R_2 \approx_\zeta R_k : \Gamma_2 \wedge \Gamma_k$ regFile
- $\triangleright C_2 \approx_\zeta C_k : \Delta_2 \wedge \Delta_k$ cstackHigh and
- $\mathbf{pc}_i \not\sqsubseteq \zeta$, for all $3 \leq i \leq k$.

Since ζ -equivalence of continuation stacks and register files is reflexive and transitive, we are left to verify that $\triangleright H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_1$ heap where $H_2 = H_1[p := h]$. For this one we must show that $\forall \ell \in \text{Dom}_\cup(\Psi_1, \Psi_2)$ the following condition holds:

$$\text{label}(\Psi_1(\ell)) \sqsubseteq \zeta \text{ or } \text{label}(\Psi_2(\ell)) \sqsubseteq \zeta, \text{ implies } \begin{cases} \ell \in \text{Dom}_\cap(H_1, H_2, \Psi_1, \Psi_2), \text{ and} \\ \Psi_1(\ell) = \Psi_2(\ell), \text{ and} \\ \triangleright_{\Psi_1, \Psi_2} H_1(\ell) \approx_\zeta H_2(\ell) : \Psi_1(\ell) \text{ hval} \end{cases}$$

If $\ell \neq p$, then H_1 and H_2 coincide and are tested at the same type $\Psi_1(p)$ and the condition is straightforward to verify. Suppose, therefore, that $\ell = p$. Since $\Gamma_1(r_d) = \Psi_1(p)$, by Relating Ψ and Γ Lemma (Lemma A.1) the label of $\Psi_1(p)$ and l are identical, we consider two cases:

- $l \not\sqsubseteq \zeta$. Then $\text{label}(\Psi_1(\ell)) \sqsubseteq \zeta$ or $\text{label}(\Psi_2(\ell)) \sqsubseteq \zeta$ is false and hence concludes the case.
- $l \sqsubseteq \zeta$. We are left to verify that $\triangleright_{\Psi_1, \Psi_1} \langle \overline{w} \rangle \approx_\zeta h : \Psi_1(p)$ wval. From $\triangleright H_1 : \Psi_1$ heap we know that $\triangleright_\Psi \langle \overline{w} \rangle : \Psi_1(p)$ hval. Thus $\triangleright_{\Psi_1} w_i : \sigma_i$ wval for each $i \in 1..n-1$ and also, by reflexivity of ζ -indistinguishability of word values $\triangleright_{\Psi_1, \Psi_1} w_i \approx_\zeta w_i : \sigma_i$ wval. Also, since $\text{label}(\sigma_i) \not\sqsubseteq \zeta$, trivially we have $\triangleright_{\Psi_1, \Psi_1} w_i \approx_\zeta \hat{R}_1(r_s) : \sigma_i$ wval. Finally, both tuples are typable and hence we may conclude the case.

□

Lemma C.3 (Prune Left) If

- $\triangleright \underline{L} \cdot C_1 \approx_\zeta C_2 : \kappa^l \cdot \Delta_1 \wedge \Delta_2$ cstackHigh and
- $\kappa = \langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\}$ with $\mathbf{pc}' \sqcup l \not\sqsubseteq \zeta$

Then $\triangleright C_1 \approx_\zeta C_2 : \Delta_1 \wedge \Delta_2$ cstackHigh. Likewise if we replace “cstackHigh” with “cstackLow” in the statement.

Proof: By simultaneous induction on the derivation of $\triangleright \underline{L} \cdot C_1 \approx_\zeta C_2 : \kappa^l \cdot \Delta_1 \wedge \Delta_2$ cstackHigh and $\triangleright \underline{L} \cdot C_1 \approx_\zeta C_2 : \kappa^l \cdot \Delta_1 \wedge \Delta_2$ cstackLow.

□

Lemma C.4 (High-pc Step) Let $\Sigma_1 = (H_1, R_1, B_1, \mathbf{pc}_1, C_1)$ and $\Sigma_2 = (H_2, R_2, B_2, \mathbf{pc}_2, C_2)$. If the following conditions hold

- $\triangleright \Sigma_1 \approx_\zeta \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2]$ program, for some heap, register and stack types Ψ_i, Γ_i and Δ_i , respectively.
- $\text{pc}_1 \not\sqsubseteq \zeta$ and $\text{pc}_2 \not\sqsubseteq \zeta$
- $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$, for some heap, register and stack types Ψ'_1, Γ'_1 and Δ'_1 , respectively.

Then either the program $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ diverges or there exists a program $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ such that

- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \xrightarrow{*} \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ and
- $\triangleright \Sigma'_1 \approx_\zeta \Sigma'_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi'_2, \Gamma'_2, \Delta'_2]$ program, for some heap, register and stack types Ψ'_2, Γ'_2 and Δ'_2 , respectively.

Proof: If $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ diverges, then we are done. Otherwise we proceed by case analysis on the reduction step $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$. Note that if the reduction step is OS_Jmp, OS_Arith, OS_Bnz1, OS_Bnz2, OS_Mov, OS_Load or OS_Store, then it suffices to pick $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2] = \Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$. Indeed, by the High-Step Invariant Lemma (Lemma 3.3):

- $\triangleright H_1 \approx_\zeta H'_1 : \Psi_1 \wedge \Psi'_1$ heap
- $\triangleright_{\Psi_1, \Psi'_1} R_1 \approx_\zeta R'_1 : \Gamma_1 \wedge \Gamma'_1$ regFile
- $\triangleright C_1 \approx_\zeta C'_1 : \Delta_1 \wedge \Delta'_1$ cstackHigh
- $\text{pc}'_1 \not\sqsubseteq \zeta$

Since $\triangleright \Sigma_1 \approx_\zeta \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2]$ program, we may use the fact that ζ -equivalence of heaps, registers and continuation stacks is an equivalence relation and deduce that:

- $\triangleright H'_1 \approx_\zeta H_2 : \Psi'_1 \wedge \Psi_2$ heap
- $\triangleright_{\Psi'_1, \Psi_2} R'_1 \approx_\zeta R_2 : \Gamma'_1 \wedge \Gamma_2$ regFile
- $\triangleright C'_1 \approx_\zeta C_2 : \Delta'_1 \wedge \Delta_2$ cstackHigh

Hence $\triangleright \Sigma'_1 \approx_\zeta \Sigma_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi_2, \Gamma_2, \Delta_2]$ program and this concludes the case.

The two remaining cases are when the reduction step $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$ is OS_Jmpcc or OS_Push. These are dealt with separately.

- OS_Jmpcc. The reduction step is

$$\frac{H_1(\underline{L}) = B \quad \kappa = \langle \text{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\}}{(H_1, R_1, \text{jmpcc}, \text{pc}_1, \underline{L} \cdot C_{11}) : [\Psi_1, \Gamma_1, \kappa^l \cdot \Delta_{11}] \mapsto (H_1, R_1, B, \text{pc}' \sqcup l, C_{11}) : [\Psi_1, \Gamma_1, \Delta_{11}]} \text{OS_Jmpcc}$$

We now consider two cases:

- $\text{pc}' \sqcup l \not\sqsubseteq \zeta$. Since $\triangleright H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_2$ heap and $\triangleright_{\Psi_1, \Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regFile hold by hypothesis we are left to verify that $\triangleright C_{11} \approx_\zeta C_2 : \Delta_{11} \wedge \Delta_2$ cstackHigh. This follows from hypothesis $\triangleright \underline{L} \cdot C_{11} \approx_\zeta C_2 : \kappa^l \cdot \Delta_{11} \wedge \Delta_2$ cstackHigh and Prune Left Lemma (Lemma C.3).
- $\text{pc}' \sqcup l \sqsubseteq \zeta$. Since $\triangleright \underline{L} \cdot C_{11} \approx_\zeta C_2 : \kappa^l \cdot \Delta_{11} \wedge \Delta_2$ cstackHigh we deduce that:
 - * $C_2 = \underline{L}_1 \cdot \dots \cdot \underline{L}_n \cdot \underline{L} \cdot C_{21}$
 - * $\Delta_2 = \kappa_1^{l_1} \cdot \dots \cdot \kappa_n^{l_n} \cdot \kappa^l \cdot \Delta_{21}$
 - * $\kappa_i = \langle \text{pc}'_i \rangle \Gamma'_i \mid \Delta'_i \rightarrow \{\}$ implies $\text{pc}'_i \sqcup l_i \not\sqsubseteq \zeta$, for all $1 \leq i \leq n$, and
 - * $\triangleright C_{11} \approx_\zeta C_{21} : \Delta_{11} \wedge \Delta_{21}$ cstackLow

Since $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ does not diverge, by Progress (Proposition 2.1) there exists a program of the form:

$$\Sigma_3 = (H_3, R_3, \text{jmpcc}, \mathbf{pc}_3, \underline{L} \cdot C_{21}) : [\Psi_3, \Gamma_3, \kappa^l \cdot \Delta_{21}]$$

such that

$$\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \xrightarrow{k} \Sigma_3 : [\Psi_3, \Gamma_3, \kappa^l \cdot \Delta_{21}]$$

and $\underline{L} \cdot C_{21}$ of type $\kappa^l \cdot \Delta_{21}$ is ζ -topped in the continuation stack of each intermediate typed machine configuration in the reduction sequence. By Subject Reduction (Proposition 2.2) $\triangleright \Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ program and we may thus apply the High-Step Invariant Lemma (Lemma 3.3) and deduce:

- * $\triangleright H_2 \approx_\zeta H_3 : \Psi_2 \wedge \Psi_3 \text{ heap}$
- * $\triangleright_{\Psi_2, \Psi_3} R_2 \approx_\zeta R_3 : \Gamma_2 \wedge \Gamma_3 \text{ regFile}$
- * $\triangleright C_2 \approx_\zeta C_3 : \Delta_2 \wedge \Delta_3 \text{ cstackHigh}$
- * the program counter \mathbf{pc} of each intermediate program in the reduction sequence satisfies $\mathbf{pc} \not\sqsubseteq \zeta$.

We now appeal to Progress (Proposition 2.1) to obtain a reduction step from $\Sigma_3 : [\Psi_3, \Gamma_3, \kappa^l \cdot \Delta_{21}]$:

$$\frac{H_3(\underline{L}) = B' \quad \kappa = \langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\}}{(H_3, R_3, \text{jmpcc}, \mathbf{pc}_3, \underline{L} \cdot C_{21}) : [\Psi_3, \Gamma_3, \kappa^l \cdot \Delta_{21}] \mapsto (H_3, R_3, B', \mathbf{pc}' \sqcup l, C_{21}) : [\Psi_3, \Gamma_3, \Delta_{21}]} \text{OS_Jmpcc}$$

Since $\triangleright H_1 \approx_\zeta H_3 : \Psi_1 \wedge \Psi_3 \text{ heap}$ we would like to deduce that $B = H_1(\underline{L}) = H_3(\underline{L}) = B'$ by using the fact that $\text{label}(\Psi_1(\underline{L})) \sqsubseteq \zeta$. However, from $\mathbf{pc}' \sqcup l \sqsubseteq \zeta$ we know that $l \sqsubseteq \zeta$ and thus we must relate l and $\text{label}(\Psi_1(\underline{L}))$. This follows from the fact that the machine configuration $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1]$ is typed: $\triangleright_{\Psi_1} \underline{L} \cdot C_{11} : \kappa^l \cdot \Delta_{11} \text{ cstack}$ holds and hence $\Psi_1(\underline{L}) \leq \kappa^l$, from which we deduce that $l = \text{label}(\Psi_1(\underline{L}))$. We conclude the case by the fact that ζ -equivalence of heaps, register files and stacks are equivalence relations.

- OS.Push. The reduction step is:

$$\frac{\mathbf{pc}_1 \sqsubseteq \text{label}(\Psi_1(\underline{L}))}{(H_1, R_1, \text{cpush } \underline{L}; B, \mathbf{pc}_1, C_1) : [\Psi_1, \Gamma_1, \Delta_1] \mapsto (H_1, R_1, B, \mathbf{pc}_1, \underline{L} \cdot C_1) : [\Psi_1, \Gamma, \Psi_1(\underline{L}) \cdot \Delta_1]} \text{OS_Push}$$

Since the heaps and register files together with their corresponding types do not change, we may reason as above and deduce that:

- $\triangleright H'_1 \approx_\zeta H_2 : \Psi'_1 \wedge \Psi_2 \text{ heap}$
- $\triangleright_{\Psi'_1, \Psi_2} R'_1 \approx_\zeta R_2 : \Gamma'_1 \wedge \Gamma_2 \text{ regFile}$

Since $\mathbf{pc}'_1 \not\sqsubseteq \zeta$ we are left to verify that $\triangleright \underline{L} \cdot C_1 \approx_\zeta C_2 : \Psi_1(\underline{L}) \cdot \Delta_1 \wedge \Delta_2 \text{ cstackHigh}$. By reflexivity of ζ -equivalence for continuation stacks we know that $\triangleright C_1 \approx_\zeta C_1 : \Delta_1 \wedge \Delta_1 \text{ cstackHigh}$. Thus we may derive:

$$\frac{\triangleright C_1 \approx_\zeta C_1 : \Delta_1 \wedge \Delta_1 \text{ cstackHigh} \quad \Psi_1(\underline{L}) = (\langle \mathbf{pc}' \rangle \Gamma' \mid \Delta' \rightarrow \{\})^l \quad \mathbf{pc}' \sqcup l \not\sqsubseteq \zeta}{\triangleright C_1 \approx_\zeta \underline{L} \cdot C_1 : \Delta_1 \wedge \Psi_1(\underline{L}) \cdot \Delta_1 \text{ cstackHigh}} \text{HighRight}$$

Note that $\mathbf{pc}' \sqcup l \not\sqsubseteq \zeta$ follows from $\mathbf{pc}_1 \not\sqsubseteq \zeta$ and the condition in the reduction step $\mathbf{pc}_1 \sqsubseteq \text{label}(\Psi_1(\underline{L}))$. Then using the fact that ζ -equivalence for continuation stacks is an equivalence relation and that $\triangleright C_1 \approx_\zeta C_2 : \Delta_1 \wedge \Delta_2 \text{ cstackHigh}$ we may conclude the case.

□

Lemma C.5 (Weaving Low and High-Pc Steps) Suppose

- $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto k\Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$
- $\triangleright \Sigma_1 \approx_\zeta \Sigma_2 : [\Psi_1, \Gamma_1, \Delta_1] \wedge [\Psi_2, \Gamma_2, \Delta_2] \text{ program}$
- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ does not diverge

Then there exists a program $\Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$ such that

- $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \mapsto^* \Sigma'_2 : [\Psi'_2, \Gamma'_2, \Delta'_2]$
- $\triangleright \Sigma'_1 \approx_\zeta \Sigma'_2 : [\Psi'_1, \Gamma'_1, \Delta'_1] \wedge [\Psi'_2, \Gamma'_2, \Delta'_2] \text{ program}$

Proof: By induction on the number of reduction steps k in the reduction sequence $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto^* \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$. If $k = 0$ then the result trivially holds. Otherwise $k = n + 1$ and

$$\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto^* \Sigma_{11} : [\Psi_{11}, \Gamma_{11}, \Delta_{11}] \mapsto \Sigma'_1 : [\Psi'_1, \Gamma'_1, \Delta'_1]$$

where n reduction steps take place in $\Sigma_1 : [\Psi_1, \Gamma_1, \Delta_1] \mapsto^* \Sigma_{11} : [\Psi_{11}, \Gamma_{11}, \Delta_{11}]$.

By the induction hypothesis there exists a program $\Sigma_{21} : [\Psi_{21}, \Gamma_{21}, \Delta_{21}]$ such that $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2] \mapsto^* \Sigma_{21} : [\Psi_{21}, \Gamma_{21}, \Delta_{21}]$ and $\triangleright \Sigma_{11} \approx_\zeta \Sigma_{21} : [\Psi_{11}, \Gamma_{11}, \Delta_{11}] \wedge [\Psi_{21}, \Gamma_{21}, \Delta_{21}] \text{ program}$. From the latter statement, we know that either:

- $\mathbf{pc}_{11} = \mathbf{pc}_{21} \sqsubseteq \zeta$, or
- $\mathbf{pc}_{11} \not\sqsubseteq \zeta$ and $\mathbf{pc}_{21} \not\sqsubseteq \zeta$

In the first case we apply the Low-Pc Step Lemma (Lemma 3.2) and conclude. In the second case we use the fact that $\Sigma_{21} : [\Psi_{21}, \Gamma_{21}, \Delta_{21}]$ does not diverge (for otherwise $\Sigma_2 : [\Psi_2, \Gamma_2, \Delta_2]$ would diverge, thus contradicting our assumption) and apply the High-Pc Step Lemma (Lemma 3.4) and conclude.

□