

CS 4110

# Programming Languages & Logics

## Lecture 36 Typed Assembly Language

26 November 2012



# Overview

---

Slogan: “Safety through types”

- An architecture for safe mobile code
  - ▶ Download annotated binaries from an untrusted code producer
  - ▶ Verify code using a trusted typechecker
  - ▶ Link and execute without errors
- Security properties hinge on understanding behavior
  - ▶ Must reason precisely about programs
  - ▶ Define “good” and “bad” behaviors
  - ▶ Identify and rule out “bad programs”
- Typed Assembly Language (TAL) is a framework that accomplishes these goals in a setting where the programs in question are x86 executables

# Schedule

---

## Today

- Typed Assembly Language
- Prelim #2 hand back

## Wednesday

- Polymorphism
- Stack Types

## Friday

- Compilation
- Course Review

# Acknowledgments

- These lectures developed by David Walker (Princeton)
- They describe *Typed Assembly Language*, a project at Cornell led by Greg Morrisett about 15 years ago
- Paper: G. Morrisett, D. Walker, K. Crary, and N. Glew. "From System F to Typed Assembly Language." In *ACM TOPLAS*. 21(3):527–568. May 1999.

## From System F to Typed Assembly Language\*

Greg Morrisett   David Walker   Karl Crary   Neal Glew  
Cornell University

### Abstract

We motivate the design of a statically typed assembly language (TAL) and present a type-preserving translation from System F to TAL. The TAL we present is based on a conventional RISC assembly language, but its static type system provides support for enforcing high-level language abstractions, such as closures, tuples, and objects, as well as user-defined abstract data types. The type system ensures that well-typed programs cannot violate these abstractions. In addition, the typing constructs place almost no restrictions on low-level optimizations such as register allocation, instruction selection, or instruction scheduling.

Our translation to TAL is specified as a sequence of type-preserving transformations, including CPS and closure conversion phases; type-correct source programs are mapped to type-correct assembly language. A key contribution is an approach to polymorphic closure conversion that is considerably simpler than previous work. The compiler and typed assembly language provide a fully automatic way to produce *proof carrying code*, suitable for use in systems where untrusted and potentially malicious code must be checked for safety before execution.

\*This material is based on work supported in part by the AFOSR grant F49620-95-1-0081, ARPA/AF grant F49620-95-1-0017, and AASERT grant N00014-95-1-0081. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

To appear at the 1998 Symposium on Principles of Programming Languages

### 1 Introduction and Motivation

Compiling a source language to a statically typed intermediate language has compelling advantages over a conventional untyped compiler. An optimizing compiler for a high-level language such as ML may make as many as 20 passes over a single program, performing sophisticated analyses and transformations such as CPS conversion [14, 35, 2, 12, 18], closure conversion [20, 40, 28, 3, 26], unboxing [22, 28, 36], inlining/exception elimination [9, 11], or region inference [7]. Many of these optimizations require type information in order to proceed, and even those that do not often benefit from the additional structure supplied by a typing discipline [22, 18, 28, 37]. Furthermore, the ability to type-check intermediate code provides an invaluable tool for debugging new transformations and optimizations [41, 30].

Today a small number of compilers work with typed intermediate languages in order to realize some or all of these benefits [22, 36, 6, 41, 24, 39, 13]. However, in all of these compilers, there is a conceptual line where types are lost. For instance, the TAL/ML compiler preserves type information through approximately 80% of compilation, but the remaining 20% is untyped.

We show how to eliminate the untyped portions of a compiler and by so doing, extend the approach of compiling with typed intermediate languages to typed target languages. The target language in this paper is a strongly typed assembly language (TAL) based on a generic RISC instruction set. The type system for the language is surprisingly standard, supporting tuples, polymorphism, mutability, and a very restricted form of function pointer, yet it is sufficiently powerful that we can automatically generate well-typed and efficient code from high-level ML-like languages. Furthermore, we claim that the type system does not seriously hinder low-level optimizations such as register allocation, instruction selection, instruction scheduling, and copy propagation.

TAL not only allows us to reap the benefits of types throughout a compiler, but it also enables a practical system for executing untrusted code both safely and

# What is TAL?

---

## In Theory

- A RISC-like assembly language
- A formal operational semantics
- A family of type systems that capture key safety properties of registers, stack, and the heap
- Rigorous proofs of soundness which demonstrate that TAL enforces security guarantees

# What is TAL?

---

## In Theory

- A RISC-like assembly language
- A formal operational semantics
- A family of type systems that capture key safety properties of registers, stack, and the heap
- Rigorous proofs of soundness which demonstrate that TAL enforces security guarantees

## In Practice

- A typechecker for almost all of the Intel IA32 architecture
- A collection of tools for assembling linking, etc. TAL binaries
- A compiler for a safe C-like language called Popcorn

# Example

High-level code:

```
fact (n,a) =  
    if (n ≤ 0) then a  
    else fact(n-1,a × n)
```

Assembly code:

```
% r1 holds n, r2 holds a, r31 holds return address  
fact:  ble r1,L2      % if n ≤ 0 goto L2  
        mul r2,r2,r1  % a := a × n  
        sub r1,r1,1    % n := n - 1  
        jmp fact      % goto fact  
  
L2:    mov r1,r2      % result := a  
        jmp r31       % return
```

# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$



# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in \textit{Identifier}$

# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in \textit{Identifier}$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$

# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in \text{Identifier}$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$
- Blocks:  $B ::= i; B \mid \text{jmp } v$

# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in Identifier$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$
- Blocks:  $B ::= i; B \mid \text{jmp } v$
- Instructions:  $i ::= aop\ r_d, r_s, v \mid bop\ r, v \mid \text{mov } r, v$

# TAL Syntax

---

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in Identifier$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$
- Blocks:  $B ::= i; B \mid \text{jmp } v$
- Instructions:  $i ::= aop\ r_d, r_s, v \mid bop\ r, v \mid \text{mov } r, v$
- Operands:  $v ::= r \mid L \mid v$

# TAL Syntax

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in Identifier$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$
- Blocks:  $B ::= i; B \mid \text{jmp } v$
- Instructions:  $i ::= aop\ r_d, r_s, v \mid bop\ r, v \mid \text{mov } r, v$
- Operands:  $v ::= r \mid L \mid v$
- Arithmetic Operations:  $aop ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \dots$

# TAL Syntax

Models a simple RISC-like assembly language.

- Registers:  $r \in \{r_1, r_2, r_3, \dots\}$
- Labels:  $L \in Identifier$
- Integers:  $n \in [-2^{k-1} \dots 2^{k-1})$
- Blocks:  $B ::= i; B \mid \text{jmp } v$
- Instructions:  $i ::= aop\ r_d, r_s, v \mid bop\ r, v \mid \text{mov } r, v$
- Operands:  $v ::= r \mid L \mid v$
- Arithmetic Operations:  $aop ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \dots$
- Branch Operations:  $bop ::= \text{beq} \mid \text{bgt} \mid \dots$

# TAL Abstract Machine

---

Model evaluation using a transition function  $\Sigma \mapsto \Sigma'$  from machine states to machine states



# TAL Abstract Machine

---

Model evaluation using a transition function  $\Sigma \mapsto \Sigma'$  from machine states to machine states

- Machine states:  $\Sigma = (H, R, B)$

# TAL Abstract Machine

---

Model evaluation using a transition function  $\Sigma \mapsto \Sigma'$  from machine states to machine states

- Machine states:  $\Sigma = (H, R, B)$
- The **heap**  $H$  is a partial map from labels  $L$  to blocks  $B$

# TAL Abstract Machine

Model evaluation using a transition function  $\Sigma \mapsto \Sigma'$  from machine states to machine states

- Machine states:  $\Sigma = (H, R, B)$
- The **heap**  $H$  is a partial map from labels  $L$  to blocks  $B$
- The **register file**  $R$  maps registers to values. Abusing notation slightly, we extend  $R$  to a map on values as follows:

$$\begin{aligned} R(n) &= n \\ R(L) &= L \\ R(r) &= v \quad \text{if } R = \{\dots, r \mapsto v, \dots\} \end{aligned}$$

# TAL Abstract Machine

Model evaluation using a transition function  $\Sigma \mapsto \Sigma'$  from machine states to machine states

- Machine states:  $\Sigma = (H, R, B)$
- The **heap**  $H$  is a partial map from labels  $L$  to blocks  $B$
- The **register file**  $R$  maps registers to values. Abusing notation slightly, we extend  $R$  to a map on values as follows:

$$\begin{aligned} R(n) &= n \\ R(L) &= L \\ R(r) &= v \quad \text{if } R = \{\dots, r \mapsto v, \dots\} \end{aligned}$$

- The **current block**  $B$  is the block associated to the (implicit) program counter

# TAL Operational Semantics (Selected Rules)

---

$$\overline{(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)}$$

# TAL Operational Semantics (Selected Rules)

$$\frac{}{(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)}$$

$$\frac{n = R(v) + R(r_s)}{(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)}$$

# TAL Operational Semantics (Selected Rules)

$$\frac{}{(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)}$$

$$\frac{n = R(v) + R(r_s)}{(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)}$$

$$\frac{R(v) = L \quad H(L) = B}{(H, R, \text{jmp } v) \mapsto (H, R, B)}$$

# TAL Operational Semantics (Selected Rules)

$$\frac{}{(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)}$$

$$\frac{n = R(v) + R(r_s)}{(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)}$$

$$\frac{R(v) = L \quad H(L) = B}{(H, R, \text{jmp } v) \mapsto (H, R, B)}$$

$$\frac{R(r) \neq 0}{(H, R, \text{beq } r, v; B) \mapsto (H, R, B)}$$



# TAL Operational Semantics (Selected Rules)

$$\frac{}{(H, R, \text{mov } r_d, v; B) \mapsto (H, R[r_d := R(v)], B)}$$

$$\frac{n = R(v) + R(r_s)}{(H, R, \text{add } r_d, r_s, v; B) \mapsto (H, R[r_d := n], B)}$$

$$\frac{R(v) = L \quad H(L) = B}{(H, R, \text{jmp } v) \mapsto (H, R, B)}$$

$$\frac{R(r) \neq 0}{(H, R, \text{beq } r, v; B) \mapsto (H, R, B)}$$

$$\frac{R(r) = 0 \quad R(v) = L \quad H(L) = B'}{(H, R, \text{beq } r, v; B) \mapsto (H, R, B')}$$

# Errors

- The machine is **stuck** if there does not exist a transition from the current state to some following state
- We will use stuck states to define the “bad” behaviors that may occur at run-time
- The type system will guarantee that well-typed machines never get stuck
- Example stuck states:
  - ▶  $(H, R, \text{add } r_d, r_s, v; B)$  where  $r_s$  and  $v$  aren't integers
  - ▶  $(H, R, \text{jmp } v)$  where  $v$  isn't a label
  - ▶  $(H, R, \text{beq } r, v)$  where  $r$  isn't an integer or  $v$  isn't a label
- To distinguish integers and labels we need a type system!

# Types

## Syntax

- $\tau ::= \text{int} \mid \Gamma \rightarrow \{\}$
- $\Gamma ::= \{r_1 : \tau_1, r_2 : \tau_2, \dots\}$

# Types

## Syntax

- $\tau ::= \text{int} \mid \Gamma \rightarrow \{\}$
- $\Gamma ::= \{r_1 : \tau_1, r_2 : \tau_2, \dots\}$

## Code Types

- Labels are like functions that take a record of arguments
- Labels have types of the form  $\{r_1 : \tau_1, r_2 : \tau_2, \dots\} \rightarrow \{\}$
- To jump to code with this type, register  $r_1$  must contain a value of type  $\tau_1$ , register  $r_2$  must contain a value of type  $\tau_2$ , and so on
- The order that register names appear is irrelevant
- Note that functions never return—every block ends with a `jmp`

# Well-Typed Example

%  $r_1$  holds  $n$ ,  $r_2$  holds  $a$ ,  $r_{31}$  holds return address

*fact*:  $\{r_1 : \text{int}, r_2 : \text{int}, r_{31} : \{r_1 : \text{int}\} \rightarrow \{\}\} \rightarrow \{\}$

ble  $r_1, L2$            % if  $n \leq 0$  goto  $L2$

mul  $r_2, r_2, r_1$        %  $a := a \times n$

sub  $r_1, r_1, 1$        %  $n := n - 1$

jmp *fact*           % goto *fact*

$L2$ :  $\{r_1 : \text{int}, r_2 : \text{int}, r_{31} : \{r_1 : \text{int}\} \rightarrow \{\}\} \rightarrow \{\}$

mov  $r_1, r_2$        % result :=  $a$

jmp  $r_{31}$        % return

# III-Typed Example

%  $r_1$  holds  $n$ ,  $r_2$  holds  $a$ ,  $r_{31}$  holds return address

*fact*:  $\{r_1 : \text{int}, r_{31} : \{r_1 : \text{int}\} \rightarrow \{\}\} \rightarrow \{\}$

ble  $r_1, L2$

mul  $r_2, r_2, r_1$       % Error!  $r_2$  doesn't have a type

sub  $r_1, r_1, 1$

jmp  $L1$       % Error! No such label

*L2*:  $\{r_2 : \text{int}, r_{31} : \{r_1 : \text{int}\} \rightarrow \{\}\} \rightarrow \{\}$

mov  $r_{31}, r_2$

jmp  $r_{31}$       % Error!  $r_{31}$  not a label

# Typechecking Overview

- Intuitively, the type system needs to keep track of:
  - ▶ The types of the registers at each point in the code
  - ▶ The types of the labels on the code
- Heap types:  $\Psi$  maps labels to code types
- Register types:  $\Gamma$  maps registers to types
- A family of typing (and subtyping) relations:
  - ▶  $\Psi; \Gamma \vdash v : \tau$
  - ▶  $\Psi \vdash i : \Gamma \rightarrow \Gamma'$
  - ▶  $\tau \leq \tau'$
  - ▶  $\vdash H : \Psi$
  - ▶  $\vdash R : \Gamma$
  - ▶  $\vdash (H, R, B)$

# Typechecking Values

---

$$\boxed{\Psi; \Gamma \vdash v : \tau}$$



# Typechecking Values

$$\boxed{\Psi; \Gamma \vdash v : \tau}$$

$$\overline{\Psi; \Gamma \vdash n : \text{int}}$$

# Typechecking Values

$$\boxed{\Psi; \Gamma \vdash v : \tau}$$

$$\overline{\Psi; \Gamma \vdash n : \text{int}}$$

$$\frac{\Gamma(r) = \tau}{\Psi; \Gamma \vdash r : \tau}$$

# Typechecking Values

$$\boxed{\Psi; \Gamma \vdash v : \tau}$$

$$\overline{\Psi; \Gamma \vdash n : \text{int}}$$

$$\frac{\Gamma(r) = \tau}{\Psi; \Gamma \vdash r : \tau}$$

$$\frac{\Psi(L) = \tau}{\Psi; \Gamma \vdash L : \tau}$$

# Subtyping

- A program won't crash if the register file has more values that are needed to satisfy the typing conditions
- Formally, a register file with more components is a subtype of a register file with fewer components:

$$\overline{\{r_1 : \tau_1, \dots, r_i : \tau_i, r_{i+1} : \tau_{i+1}\}} \leq \overline{\{r_1 : \tau_1, \dots, r_i : \tau_i\}}$$

Note that this is the ordinary rule for records!

- Code subtyping goes in the opposite direction: a label requiring  $r_1$  and  $r_2$  may be used as a label requiring  $r_1, r_2$ , and  $r_3$ .

$$\frac{\Gamma' \leq \Gamma}{\Gamma \rightarrow \{\} \leq \Gamma' \rightarrow \{\}}$$

Note that this is the ordinary *contravariant* rule for functions!

# Subtyping

- Subtyping is also reflexive and transitive.

$$\frac{}{\tau \leq \tau}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

- A subsumption rule allows values to be used at supertypes:

$$\frac{\Psi; \Gamma \vdash v : \tau_1 \quad \tau_1 \leq \tau_2}{\Psi; \Gamma \vdash v : \tau_2}$$

# Typing Instructions

$$\boxed{\Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2}$$

- $\Gamma_1$  describes the registers before the execution of the instruction—a *precondition*
- $\Gamma_2$  describes the registers after the execution of the instruction—a *postcondition*
- $\Psi$  is invariant. That is, the types of objects on the heap will not change (at least for now...)

# Typing Instructions

$$\boxed{\Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2}$$

## Arithmetic operations

$$\frac{\Psi; \Gamma \vdash r_s : \text{int} \quad \Psi; \Gamma \vdash v : \text{int}}{\Psi \vdash \text{aop } r_d, r_s, v : \Gamma \rightarrow \Gamma[r_d := \text{int}]}$$

## Conditional branches

$$\frac{\Psi; \Gamma \vdash r : \text{int} \quad \Psi; \Gamma \vdash v : \Gamma \rightarrow \{\}}{\Psi \vdash \text{bop } r, v : \Gamma \rightarrow \Gamma}$$

## Data movement

$$\frac{\Psi; \Gamma \vdash v : \tau}{\Psi \vdash \text{mov } r_d, v : \Gamma \rightarrow \Gamma[r_d := \tau]}$$

# Typing Instructions

$$\boxed{\Psi \vdash i : \Gamma_1 \rightarrow \Gamma_2}$$

## Jumps

$$\frac{\Psi; \Gamma \vdash v : \Gamma \rightarrow \{\}}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

## Basic blocks

$$\frac{\Psi; \Gamma \vdash i : \Gamma_1 \rightarrow \Gamma_2 \quad \Psi; \Gamma \vdash B : \Gamma_2 \rightarrow \{\}}{\Psi \vdash i; B : \Gamma_1 \rightarrow \{\}}$$



# Heap, Register File, and Machine Typing

## Heaps

$$\frac{\text{dom}(H) = \text{dom}(\Psi) \quad \forall L \in \text{dom}(H). \Psi \vdash H(L) : \Psi(L)}{\vdash H : \Psi}$$

## Register Files

$$\frac{\forall r \in \text{dom}(\Gamma). \Psi; \{\} \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma}$$

## Machines

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash B : \Gamma \rightarrow \{\}}{\vdash (H, R, B)}$$

# Type Safety

The type system satisfies the following theorem:

## Theorem (Type Safety)

*If  $\vdash \Sigma$  and  $\Sigma \mapsto^* \Sigma'$ , then  $\Sigma'$  is not stuck.*

# Type Safety

The type system satisfies the following theorem:

## Theorem (Type Safety)

*If  $\vdash \Sigma$  and  $\Sigma \mapsto^* \Sigma'$ , then  $\Sigma'$  is not stuck.*

Proof:

- Progress: if a state is well-typed, then it is not stuck
- Preservation: evaluation preserves types

# Type Safety

The type system satisfies the following theorem:

## Theorem (Type Safety)

*If  $\vdash \Sigma$  and  $\Sigma \mapsto^* \Sigma'$ , then  $\Sigma'$  is not stuck.*

Proof:

- Progress: if a state is well-typed, then it is not stuck
- Preservation: evaluation preserves types

## Corollary

- *Every jump in a well-typed program is to a valid label*
- *Every arithmetic operation in a well-typed program is done with integers—not labels!*

# Canonical Forms

## Lemma

If  $\vdash H : \Psi$  and  $\Psi \vdash R : \Gamma$  and  $\Psi; \Gamma \vdash v : \tau$  then

- $\tau = \text{int}$  implies  $R(v) = n$
- $\tau = \{r_1 : \tau_1, \dots, r_k : \tau_k\} \rightarrow \{\}$  implies  $R(v) = L$ .

Moreover  $H(L) = B$  and  $\Psi \vdash B : \{r_1 : \tau_1, \dots, r_k : \tau_k\} \rightarrow \{\}$

Proof: by induction on typing derivations...

# Progress (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  then there exists a  $\Sigma_2$  such that  $\Sigma_1 \mapsto \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

# Progress (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  then there exists a  $\Sigma_2$  such that  $\Sigma_1 \mapsto \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

The third premise must be a derivation that ends in the rule:

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

# Progress (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  then there exists a  $\Sigma_2$  such that  $\Sigma_1 \mapsto \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

The third premise must be a derivation that ends in the rule:

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

By Canonical Forms, we have  $R(v) = L$  and  $H(L) = B'$ .



# Progress (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  then there exists a  $\Sigma_2$  such that  $\Sigma_1 \mapsto \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

The third premise must be a derivation that ends in the rule:

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

By Canonical Forms, we have  $R(v) = L$  and  $H(L) = B'$ . Therefore:

$$\frac{R(v) = L \quad H(L) = B'}{(H, R, \text{jmp } v) \mapsto (H, R, B')}$$

# Preservation (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  and  $\Sigma_1 \mapsto \Sigma_2$  then  $\vdash \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

# Preservation (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  and  $\Sigma_1 \mapsto \Sigma_2$  then  $\vdash \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

The third premise must be a derivation that ends in the rule:

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

# Preservation (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  and  $\Sigma_1 \mapsto \Sigma_2$  then  $\vdash \Sigma_2$*

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}{\vdash (H, R, \text{jmp } v)}$$

The third premise must be a derivation that ends in the rule:

$$\frac{\Psi; \Gamma \vdash v : \Gamma}{\Psi \vdash \text{jmp } v : \Gamma \rightarrow \{\}}$$

Moreover, the operational rule must be

$$\frac{R(v) = L \quad H(L) = B'}{(H, R, \text{jmp } v) \mapsto (H, R, B')}$$

# Preservation (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  and  $\Sigma_1 \mapsto \Sigma_2$  then  $\vdash \Sigma_2$*

By Canonical Forms, we have  $\Psi \vdash B : \Gamma \rightarrow \{\}$

# Preservation (jmp Case)

## Lemma

*If  $\vdash \Sigma_1$  and  $\Sigma_1 \mapsto \Sigma_2$  then  $\vdash \Sigma_2$*

By Canonical Forms, we have  $\Psi \vdash B : \Gamma \rightarrow \{\}$

Therefore:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi \vdash B : \Gamma \rightarrow \{\}}{\vdash (H, R, B)}$$