Automatic reverse engineering for formal verification

Magnus O. Myreen

University of Cambridge, UK

Presented at HCSS 2009

Trust.

Do you trust your programs? ... written in C, C++, Java, Haskell

High assurance requires proof, but what is assumed about:

- the source language?
- the compiler?
- the execution environment of the target languages?

Most verification proof are of source code, but source code is not what runs on real hardware.

Trust the machine code

For hardware, programs are machine code:

```
34 F8 45 E5 34 82 03 00 ...
```

Real guarantees for actual executable code requires proving properties of machine code.

This talk:

Part 1: verification of existing machine code (via decompilation)

Part 2: construction of correct machine code (via compilation)

Part 3: case study: verified LISP interpreter

Verification of machine code

Challenges:

- machine code operates at a low level of abstraction
- machine languages differ from each other

Verification of machine code

Challenges:

- machine code operates at a low level of abstraction
- machine languages differ from each other
- detailed models of such are large and hard to learn

Verification of machine code

Challenges:

- machine code operates at a low level of abstraction
- machine languages differ from each other
- detailed models of such are large and hard to learn

machine code

$\boxed{\textit{code}}$

correctness statement

$\{P\}\ \textit{code}\ \{Q\}$

Verification of machine code

Challenges:

- machine code operates at a low level of abstraction
- machine languages differ from each other
- detailed models of such are large and hard to learn

ARM/x86/PowerPC model

machine code

$\boxed{\textit{code}}$

$\vdots$

(7800/4500/2100 lines)

$\vdots$

correctness statement

$\{P\}\ \textit{code}\ \{Q\}$

Verification of machine code

Challenges:

- machine code operates at a low level of abstraction
- machine languages differ from each other
- detailed models of such are large and hard to learn

ARM/x86/PowerPC model

machine code

$\boxed{\textit{code}}$

⋮

(7800/4500/2100 lines)

⋮

correctness statement

$\{P\}\ \textit{code}\ \{Q\}$

Contribution: a method/tool which

- exposes as little as possible of the big models to the user;
- makes non-automatic proofs independent of the models

Decompilation

Example: Given some ARM machine code,

```
 0: E3A00000
 4: E3510000
 8: 12800001
12: 15911000
16: 1AFFFFFB
```

Decompilation

Example: Given some ARM machine code,

```
 0: E3A00000      mov r0, #0
 4: E3510000   L: cmp r1, #0
 8: 12800001      addne r0, r0, #1
12: 15911000      ldrne r1, [r1]
16: 1AFFFFFB      bne L
```

Decompilation

Example: Given some ARM machine code,

```
 0: E3A00000       mov r0, #0
 4: E3510000    L: cmp r1, #0
 8: 12800001       addne r0, r0, #1
12: 15911000       ldrne r1, [r1]
16: 1AFFFFFB       bne L
```

the decompiler extracts a readable function:

$$
\begin{aligned}
f(r_0, r_1, m) &= \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m) \\
g(r_0, r_1, m) &= \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else} \\
&\qquad\quad \text{let } r_0 = r_0{+}1 \text{ in} \\
&\qquad\quad \text{let } r_1 = m(r_1) \text{ in} \\
&\qquad\qquad g(r_0, r_1, m)
\end{aligned}
$$

Decompilation, correct?

Decompiler automatically proves a certificate theorem which states that f describes the effect of the ARM code, informally:

> for any initially value (r_0, r_1, m) in reg 0, reg 1 and memory,
> the code terminates with $f(r_0, r_1, m)$ in reg 0, reg 1 and memory.

The formal HOL theorem:

$$
\begin{array}{l}
f_{pre}(r_0, r_1, m) \Rightarrow \\
\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } (r_0, r_1, m) * \mathsf{PC}\ p * \mathsf{S} \,\} \\
p : \texttt{E3A00000 E3510000 12800001 15911000 1AFFFFFB} \\
\{\, (\mathsf{R0}, \mathsf{R1}, \mathsf{M}) \text{ is } f(r_0, r_1, m) * \mathsf{PC}\ (p + 20) * \mathsf{S} \,\}
\end{array}
$$

Certificate theorems are proved automatically in the HOL4 system.

Decompilation, under the hood

The decompiler automatically derived *f* from Fox's 7800-line ARM model:

```
⋮

|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0xE3A00000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
   (ARM_WRITE_UNDEF F (ARM_WRITE_REG 0w 0w (ARM_WRITE_UNDEF F state))))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0xE3510000w) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
   (ARM_WRITE_STATUS (word_msb (ARM_READ_REG 1w state),ARM_READ_REG 1w state = 0w,
    0w <=+ ARM_READ_REG 1w state,F) (ARM_WRITE_UNDEF F state)))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x12800001w) ∧
   (¬ARM_READ_STATUS sZ state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
   (ARM_WRITE_UNDEF F (ARM_WRITE_REG 0w (ARM_READ_REG 0w state + 1w) (ARM_WRITE_UNDEF F state))))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x12800001w) ∧
   ¬(¬ARM_READ_STATUS sZ state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F state))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x15911000w) ∧
   (¬ARM_READ_STATUS sZ state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_UNDEF F (ARM_WRITE_REG 1w (FORMAT UnsignedWord ((1 >< 0)
   (ARM_READ_REG 1w state)) (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 1w state)) state))
   (ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F state))))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x15911000w) ∧
   ¬(¬ARM_READ_STATUS sZ state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F state))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x1AFFFFFBw) ∧
   (¬ARM_READ_STATUS sZ state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 0xFFFFFFF4w)
   (ARM_WRITE_UNDEF F state)),

⋮
```

Decompilation, verification example

Decompiler automatically produced: f, f_{pre} and a certificate.

- decompilation dealt with the detailed machine model
- safety preconditions were collected in f_{pre}
- user is left to do a simple manual proof

Decompilation, verification example

Decompiler automatically produced: f, f_{pre} and a certificate.

- decompilation dealt with the detailed machine model
- safety preconditions were collected in f_{pre}
- user is left to do a simple manual proof

Let *list* formalise "a linked-list is in memory":

$$\begin{array}{rcl} \mathit{list}(\mathrm{nil}, a, m) & = & a = 0 \\ \mathit{list}(\mathrm{cons}\ x\ l, a, m) & = & \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\ & & \quad \mathit{list}(l, a', m) \wedge \mathit{aligned}(a) \end{array}$$

Manual part of verification proof (14 lines):

$$\begin{array}{rcl} \forall x\ l\ a\ m.\ \ \mathit{list}(l, a, m) & \Rightarrow & f(x, a, m) = (\mathit{length}(l), 0, m) \\ \forall x\ l\ a\ m.\ \ \mathit{list}(l, a, m) & \Rightarrow & f_{pre}(x, a, m) \end{array}$$

Decompilation, verification example, cont.

Properties proved for the extracted function f carry over to properties of the machine code:

Decompilation, verification example, cont.

Properties proved for the extracted function f carry over to properties of the machine code:

$$list(l, r_1, m) \Rightarrow$$

$$\{\,(\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } (r_0, r_1, m) * \mathrm{PC}\ p * \mathrm{S}\,\}$$

p : `E3A00000 E3510000 12800001 15911000 1AFFFFFB`

$$\{\,(\mathrm{R0}, \mathrm{R1}, \mathrm{M}) \text{ is } (length(l), 0, m) * \mathrm{PC}\ (p + 20) * \mathrm{S}\,\}$$

Proof reuse

The manual proof was completely independent of the ARM model.

$\Rightarrow$ possible proof reuse!

Proof reuse

The manual proof was completely independent of the ARM model.

$\Rightarrow$ possible proof reuse!

Example

Given similar x86 and PowerPC code:

```
31C085F67405408B36EBF7
```

```
38A000002C14000040820010 7E80A02E38A500014BFFFFF0
```

which decompiles into f' and f'', respectively.

Manual proofs can be reused, if $f = f' = f''$.

Proof reuse, cont.

Decompiling the x86 code produces:

$$
\begin{array}{rcl}
f'(eax, esi, m) & = & \text{let } eax = eax \otimes eax \text{ in } g'(eax, esi, m) \\
g'(eax, esi, m) & = & \text{if } esi \,\&\, esi = 0 \text{ then } (eax, esi, m) \text{ else} \\
& & \quad \text{let } esi = m(esi) \text{ in} \\
& & \quad \text{let } eax = eax{+}1 \text{ in} \\
& & \quad\quad g'(eax, esi, m)
\end{array}
$$

But in this case, easy to prove $f = f'$ (4 lines),

- some tricks can be undone by rewriting, e.g. $\forall x.\ x \,\&\, x = x$
- resources can be renamed, e.g. substitute r_1 for eax
- some instruction orders are irrelevant, e.g. by let-expansion

Summary of part 1: decompilation

Decompilation:

- given machine code, produces HOL function + certificate
- automates all machine-specific proofs (w/o code annotations)
- proof reuse possible, in certain cases

Summary of part 1: decompilation

Decompilation:

- given machine code, produces HOL function + certificate
- automates all machine-specific proofs (w/o code annotations)
- proof reuse possible, in certain cases

Implementation:

- concise certificate theorems using separation logic
- special loop rule introduces tail-recursive functions
- robust, heuristics only used for control-flow discovery

Summary of part 1: decompilation

Decompilation:

- given machine code, produces HOL function + certificate
- automates all machine-specific proofs (w/o code annotations)
- proof reuse possible, in certain cases

Implementation:

- concise certificate theorems using separation logic
- special loop rule introduces tail-recursive functions
- robust, heuristics only used for control-flow discovery

Supported machine languages:

- ARM model by Fox [TPHOLs'03]
- x86 model by Sarkar et al. [POPL'09]
- PowerPC model by Leroy [POPL'06]

This talk:

Part 1: verification of existing machine code (via decompilation)

Part 2: construction of correct machine code (via compilation)

Part 3: case study: verified LISP interpreter

Compilation motivation

Work-flow:

1. user defines functions f, i.e. writes:

   $$\texttt{mcDefine}\ `f = \ldots'$$

2. compiler (`mcDefine`) produces machine code, which implements f, and proves a certificate theorem:

   $$\vdash \text{“the generated code executes } f\text{”}$$

3. user proves properties of f, since properties of f also describe the generated machine code.

Compilation example

Given function f as input

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

the compiler generates ARM machine code:

```
E351000A     L:  cmp r1,#10
2241100A         subcs r1,r1,#10
2AFFFFFC         bcs L
```

Compilation example

Given function f as input

$$f(r_1) \;=\; \text{if } r_1 < 10 \text{ then } r_1 \text{ else let } r_1 = r_1 - 10 \text{ in } f(r_1)$$

the compiler generates ARM machine code:

```
E351000A     L:  cmp r1,#10
2241100A         subcs r1,r1,#10
2AFFFFFC         bcs L
```

and automatically proves a certificate HOL theorem, which states that f is executed by the generated machine code:

$$\vdash \;\; \{\,\mathrm{R1}\; r_1 * \mathrm{PC}\; p * \mathrm{s}\,\}$$
$$p : \texttt{E351000A 2241100A 2AFFFFFC}$$
$$\{\,\mathrm{R1}\; f(r_1) * \mathrm{PC}\; (p{+}12) * \mathrm{s}\,\}$$

Compilation, under the hood

The compiler proved the certificate w.r.t. Fox's 7800-line ARM model:

```
.
.
.
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0xE351000Aw) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w)
   (ARM_WRITE_STATUS (word_msb (ARM_READ_REG 1w state + 0xFFFFFFF6w),
   ARM_READ_REG 1w state + 0xFFFFFFF6w = 0w, 10w <=+ ARM_READ_REG 1w state,
   word_msb (ARM_READ_REG 1w state) ∧
   (word_msb (ARM_READ_REG 1w state) <=/=> word_msb (ARM_READ_REG 1w state + 0xFFFFFFF6w)))
   (ARM_WRITE_UNDEF F state)))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x2241100Aw) ∧
   (ARM_READ_STATUS sC state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F
   (ARM_WRITE_REG 1w (ARM_READ_REG 1w state + 0xFFFFFFF6w) (ARM_WRITE_UNDEF F state)))),
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x2241100Aw) ∧
   ¬(ARM_READ_STATUS sC state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F state))
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x2AFFFFFCw) ∧
   (ARM_READ_STATUS sC state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 0xFFFFFFF8w)
   (ARM_WRITE_UNDEF F state)),
|- (ARM_READ_MEM ((31 >< 2) (ARM_READ_REG 15w state)) state = 0x2AFFFFFCw) ∧
   ¬(ARM_READ_STATUS sC state) ∧ ¬state.undefined ⇒
   (NEXT_ARM_MMU cp state = ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) (ARM_WRITE_UNDEF F state))
.
.
.
```

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \text{ mod } 10$$

Here 'mod' is modulus over unsigned machine words.

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \bmod 10$$

Here 'mod' is modulus over unsigned machine words.

Properties proved of f translate to properties of the machine code:

$$\begin{array}{ll} \vdash & \{\mathsf{R1}\ r_1 * \mathsf{PC}\ p * \mathsf{s}\} \\ & \quad p : \texttt{E351000A 2241100A 2AFFFFFC} \\ & \{\mathsf{R1}\ (r_1\ \mathsf{mod}\ 10) * \mathsf{PC}\ (p{+}12) * \mathsf{s}\} \end{array}$$

Compilation example, cont.

One can prove properties of f since it lives inside HOL:

$$\vdash \ \forall x.\ f(x) = x \text{ mod } 10$$

Here 'mod' is modulus over unsigned machine words.

Properties proved of f translate to properties of the machine code:

$$\begin{array}{ll} \vdash & \{\mathrm{R1}\ r_1 * \mathrm{PC}\ p * \mathrm{s}\} \\ & \quad p : \texttt{E351000A 2241100A 2AFFFFFC} \\ & \{\mathrm{R1}\ (r_1 \text{ mod } 10) * \mathrm{PC}\ (p{+}12) * \mathrm{s}\} \end{array}$$

Additional feature: the compiler can use the above theorem to extend its input language with: let $r_1 = r_1$ mod 10 in _

Additional feature: user-defined extensions

Using our theorem about mod, the compiler accepts:

$$
\begin{aligned}
g(r_1, r_2, r_3) \;=\; & \text{let } r_1 = r_1 + r_2 \text{ in} \\
& \text{let } r_1 = r_1 + r_3 \text{ in} \\
& \text{let } r_1 = r_1 \text{ mod } 10 \text{ in} \\
& \quad (r_1, r_2, r_3)
\end{aligned}
$$

The generated code becomes:

```
E0811002     add r1,r1,r2
E0811003     add r1,r1,r3
E351000A     MACRO INSERT r1_mod_10 [part:1/3]
2241100A     MACRO INSERT r1_mod_10 [part:2/3]
2AFFFFFC     MACRO INSERT r1_mod_10 [part:3/3]
```

Previously proved theorems can be used as building blocks for subsequent compilations.

Implementation

To compile function f:

1. code generation:
   generates, without proof, machine code from input f;
2. decompilation:
   derives, via proof, a function f' describing the machine code;
3. certification:
   proves $f = f'$.

Features:

- code generation completely separate from proof
- supports many light-weight optimisations without any additional proof burden: instruction reordering, conditional execution, dead-code elimination, duplicate-tail elimination, ...
- allows for significant user-defined extensions

This talk:

Part 1: verification of existing machine code (via decompilation)

Part 2: construction of correct machine code (via compilation)

Part 3: case study: verified LISP interpreter

Case study: verified LISP interpreter, idea

Why verify a LISP interpreter?

- simplest prototype of a complete implementation of a functional language
- provides a logically clean platform for future work
- shows that compilation scales

Builds on:

- extensible compilation from previous section
- Mike Gordon's clean relational semantics of evaluation in an applicative subset of LISP 1.5 [ACL2 workshop 2007]

The result is code which seems to be the first formally verified end-to-end implementation of a functional programming language.

Case study: verified LISP interpreter, idea

Key idea: if one shows that the ARM instruction

```
E5933000        ldr r3,[r3]
```

implements car over a heap of s-expressions (lisp):

$$
\begin{array}{l}
\text{isPair } v_1 \;\Rightarrow \\
\{\ \text{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc}\ p\ \} \\
\ p : \texttt{E5933000} \\
\{\ \text{lisp}\ (\text{car}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \text{pc}\ (p + 4)\ \}
\end{array}
$$

then the compiler is able to handle, car over s-expressions:

$$\text{let } v_1 = \text{car } v_1 \text{ in } \_$$

The compiler's user-defined extensions can handle abstraction.

Case study: verified LISP interpreter, method

A verified LISP evaluator was constructed:

1. the compiler was augmented with car, cdr, cons, etc.
2. a function lisp_eval was compiled
3. lisp_eval was proved to implement Gordon's relational semantics of evaluation in (an applicative subset of) McCarthy's LISP 1.5

As part of this, machine code was verified for:

- memory allocation and garbage collection
- parsing of s-expressions
- printing of s-expressions

Case study: verified LISP interpreter, theorem

The result is an interpreter which parses, evaluates and prints LISP.

The theorem certifying its correctness is:

$$
\begin{array}{l}
\forall s\ r\ l\ p. \\
\quad s \longrightarrow_{eval} r \ \wedge\ \mathrm{sexp\_ok}\ s \ \wedge\ \mathrm{lisp\_eval\_pre}(s, l) \implies \\
\quad \{\ \exists a.\ \mathrm{R3}\ a * \mathrm{string}\ a\ (\mathrm{sexp2string}\ s) * \mathrm{space}\ s\ l * \mathrm{pc}\ p\ \} \\
\quad\ p : \ldots \text{ machine code not shown } \ldots \\
\quad \{\ \exists a.\ \mathrm{R3}\ a * \mathrm{string}\ a\ (\mathrm{sexp2string}\ r) * \mathrm{space}'\ s\ l * \mathrm{pc}\ (p{+}8968)\ \}
\end{array}
$$

where:

$s \longrightarrow_{eval} r$	is	"s evaluates to r in Gordon's semantics"
$\mathrm{sexp\_ok}\ s$	is	"s contains no bad symbols"
$\mathrm{lisp\_eval\_pre}(s, l)$	is	"s can be evaluated with heap limit l"
$\mathrm{string}\ a\ str$	is	"string str is stored in memory at address a"
$\mathrm{space}\ s\ l$	is	"there is enough memory to setup heap of size l"

Case study: verified LISP interpreter, in use

Example: prove

$$\forall x. \quad (\texttt{prog}\ x) \rightarrow_{eval} encrypt(x)$$

then instantiate correctness theorem to show that the interpreter always computes $encrypt(x)$ when $(\texttt{prog}\ x)$ is evaluated:

$\forall x\ l\ p.$

$\quad \text{sexp\_ok x} \wedge \text{lisp\_eval\_pre}((\texttt{prog}\ x), l) \implies$

$\quad \{\ \exists a.\ \text{R3}\ a * \text{string}\ a\ (\text{sexp2string}\ (\texttt{prog}\ x)) * \text{space}\ (\texttt{prog}\ x)\ l * \text{pc}\ p\ \}$

$\quad p : \ldots$ machine code not shown ...

$\quad \{\ \exists a.\ \text{R3}\ a * \text{string}\ a\ (\text{sexp2string}\ (encrypt(x))) * \text{space}'\ l * \text{pc}\ (p{+}8968)\ \}$

Talk summary

This talk presented tools for:

- verification of machine code (decompilation) [FMCAD'08]
- construction of correct code (compilation) [CC'09]

and showed how formally verified applications can be developed:

- verified LISP eval for ARM, x86 and PowerPC [TPHOLs'09]

Ack. I thank J Moore for suggesting the phrase "automatic reverse engineering".
For details also see my dissertation: Formal verification of machine-code programs

Talk summary

This talk presented tools for:

- verification of machine code (decompilation) [FMCAD'08]
- construction of correct code (compilation) [CC'09]

and showed how formally verified applications can be developed:

- verified LISP eval for ARM, x86 and PowerPC [TPHOLs'09]

Questions?
(I'm happy to explain technical details and give a demo separately.)

Ack. I thank J Moore for suggesting the phrase "automatic reverse engineering".
For details also see my dissertation: Formal verification of machine-code programs

Extra slide: Gordon's LISP semantics

Defined using three mutually recursive relations $\rightarrow_{eval}$, $\rightarrow_{app}$ and $\rightarrow_{eval\_list}$.

$$\frac{ok\_name\ v}{(v, \rho) \rightarrow_{eval} \rho(v)} \qquad \frac{}{(c, \rho) \rightarrow_{eval} c} \qquad \frac{}{([\,], \rho) \rightarrow_{eval} \texttt{nil}}$$

$$\frac{(p, \rho) \rightarrow_{eval} \texttt{nil} \wedge ([gl], \rho) \rightarrow_{eval} s}{([p \rightarrow e; gl], \rho) \rightarrow_{eval} s} \qquad \frac{(p, \rho) \rightarrow_{eval} x \wedge x \neq \texttt{nil} \wedge (e, \rho) \rightarrow_{eval} s}{([p \rightarrow e; gl], \rho) \rightarrow_{eval} s}$$

$$\frac{can\_apply\ k\ args}{(k, args, \rho) \rightarrow_{app} k\ args} \qquad \frac{(\rho(f), args, \rho) \rightarrow_{app} s \wedge ok\_name\ f}{(f, args, \rho) \rightarrow_{app} s}$$

$$\frac{(e, \rho[args/vars]) \rightarrow_{eval} s}{(\lambda[[vars]; e], args, \rho) \rightarrow_{app} s} \qquad \frac{(fn, args, \rho[fn/x]) \rightarrow_{app} s}{(label[[x]; fn], args, \rho) \rightarrow_{app} s}$$

$$\frac{}{([\,], \rho) \rightarrow_{eval\_list} [\,]} \qquad \frac{(e, \rho) \rightarrow_{eval} s \wedge ([el], \rho) \rightarrow_{eval\_list} sl}{([e; el], \rho) \rightarrow_{eval\_list} [s; sl]}$$

Here c, v, k and f range over value constants, value variables, function constants and function variables, respectively.