The Correctness of a Code Generator for a Functional Language

Nathanaël Courant

September 8, 2017
Context

- PVS: interactive proof assistant
- Specification language almost completely executable
- Can be used to write and formally verify programs
- Slow → generate efficient executable code (PVS2C)
- Objective: formally verify PVS2C
- Idealized version
- Related work: CompCert, CakeML
A Small Functional Language
Evaluation with Reference Counting
A Small Imperative Language
Formalization in PVS
Expressions and contexts

\begin{align*}
\text{let } a &= t[i] \text{ in let } b = t[j] \text{ in let } t' = t[i \mapsto b] \text{ in } t'[j \mapsto a] \\
K ::= &\quad \square \mid \text{pop}(K_1) \\
e ::= &\quad n \mid x \\
&\quad \text{nil} \\
&\quad f(x_1, \ldots, x_n) \\
&\quad \text{let } (x : \text{int}^{*^n}) = e_1 \text{ in } e_2 \\
&\quad \text{ifnz } x \text{ then } e_1 \text{ else } e_2 \\
&\quad x[y] \\
&\quad x[y \mapsto z] \\
&\quad \text{newint}(n) \mid \text{newref}(n) \\
&\quad \text{pop}(e_1) \mid \text{ref}(k) \\

v ::= &\quad n \mid \text{nil} \mid \text{ref}(k) \\

r ::= &\quad x \\

f(x_1, \ldots, x_n) \\
x[y] \\
x[y \mapsto z] \\
\text{newint}(n) \mid \text{newref}(n) \\
\text{let } x = v \text{ in } e \\
\text{pop}(v)
\end{align*}
Decomposition theorem

- Fill context with expression: replace hole \(\square\) by expression, denoted \(K e\)
- \(K = \text{let } x = \square \text{ in } f(x), \ e = 42 \rightarrow K e = \text{let } x = 42 \text{ in } f(x)\)

Theorem (Decomposition theorem)

*If \(e\) is not a value, there exists a unique decomposition \(e = Kr\) with \(K\) a context, \(r\) a redex.*
Evaluation state

\[(e, S, M)\]

- \(e\) an expression,
- \(S\) the stack: maps variables \(\rightarrow\) values,
- \(M\) the store: maps references \(\rightarrow\) arrays of values.
Reduction of $FL$

- Defined on redxes
- Context-preserving: if $(e, S, M) \rightarrow (e', S', M')$, then $(Ke, S, M) \rightarrow (Ke', S', M')$
- Deterministic

Exemples:

$$(x, S, M) \rightarrow (S(x), S, M)$$
$$(x[y \mapsto z], S, M) \rightarrow (\text{new}(M), S, M[\text{new}(M) \mapsto M(S(x))[S(y) \mapsto S(z)])]$$
$$(\text{let } x = v \text{ in } e, S, M) \rightarrow (\text{pop}(e), \text{push}(x, v, S), M)$$
A Small Functional Language
Evaluation with Reference Counting
A Small Imperative Language
Formalization in PVS
Reference counts

- Count number of times each reference appears
- \(#(S, x)\): number of times \(x\) appears in \(S\)
- Invariant:

\[
\mathcal{C}(\text{ref}(k)) = 1_{\text{ref}(k) \in e} + #(S, \text{ref}(k)) + \sum_{\text{ref}(s) \in M} #(\mathcal{M}(\text{ref}(s)), \text{ref}(k))
\]

- Keep track of reference counts
- Free memory when count becomes 0
- Destructive updates when possible
Marked variables

- Variables can be marked + new constructor release
- Mark last occurrence of variables in each execution path
- Done statically: \( \text{mark}(X, e) \) marks each variable in \( e \) that is not in \( X \)

\[
\text{mark}(X, x) = \begin{cases} 
x & \text{if } x \in X \\
x & \text{otherwise}
\end{cases}
\]

\[
\text{mark}(X, \text{let } x = e_1 \text{ in } e_2) = \\
\begin{cases} 
\text{let } x = \text{mark}(X \cup \text{vars}(e_2), e_1) & \text{if } x \in \text{vars}(e_2) \\
in \text{mark}(X, e_2) \\
\text{let } x = \text{mark}(X \cup \text{vars}(e_2), e_1) & \text{otherwise} \\
in \text{release}(x, \text{mark}(X, e_2))
\end{cases}
\]

- Example:

\[
\text{let } x = f(y) \text{ in ifnz } z \text{ then } g(x, y) \text{ else release}(y, f(x))
\]
Invariants preserved, with states \((e, S, M)\) and \((e', S', M', C')\):

- Reference count is accurate
- A variable no longer live in \(e'\) is not bound to a reference in \(S'\)
- The expression \(e'\) is correctly marked
- All subterms \(\text{release}(x, e_2)\) of \(e'\) have \(x\) marked
- There is a function \(f\) that maps the references in \(M'\) with count \(> 0\) to references in \(M\) so that:
  - \(e\) is obtained by removing \(\text{release}\) and unmarking variables in \(f(e')\),
  - For each variable \(x\) live in \(e'\), \(S(x) = f(S'(x))\),
  - For each reference \(s\) in \(M'\) with \(C'(s) > 0\), \(M(f(s)) = f(M'(s))\)
A Small Functional Language
Evaluation with Reference Counting

A Small Imperative Language
Formalization in PVS
Syntax

\{\text{int } z; \{z := x[y]; \text{result := } x[z]\}\}

\begin{align*}
s &::= | x := e \\
e &::= | n \\
& | x \\
& | \text{nil} \\
& | f(x_1, \ldots, x_n) \\
& | x[y] \\
& | x[y \mapsto z] \\
& | \text{newint}(n) \\
& | \text{newref}(n)
\end{align*}

\begin{align*}
dcl &::= \text{int}^* x \\
\text{function} &::= (\text{name, decl}^*, s) \\
\text{program} &::= \text{function}^*
\end{align*}
Evaluation state

Fixed program

\((\mathcal{R}, S, \mathcal{M}, \mathcal{C})\)

- \(S\) is the stack,
- \(\mathcal{M}\) is the store,
- \(\mathcal{C}\) is the reference count,
- \(\mathcal{R}\) is the call stack: stack of (function, program counter, local depth)
Reduction of IL

- Extract statement at current program counter
- Reduce statement
- For assignments, except calls: like $RL$ but store result
Return variable: \texttt{translate}(e, x) sets x to result of e

\[
\text{translate}(y[z \mapsto w], x) = x := y[z \mapsto w]
\]

\[
\text{translate}(\text{ifnz } y \text{ then } e_1 \text{ else } e_2, x) =
\]

\[
\text{ifnz } y \text{ then } \text{translate}(e_1, x) \text{ else } \text{translate}(e_2, x)
\]

\[
\text{translate}(\text{let } (y : \text{int}^* n) = e_1 \text{ in } e_2, x) =
\]

\[
\{ \text{int}^* n \ y; \{ \text{translate}(e_1, y); \{ \text{skip}; \text{translate}(e_2, x) \} \} \}
\]
Correctness

- Reconstruct redex and context from call stack,
- Reconstruct mapping for the stack variables,
- Store and counts the same
Program

\begin{align*}
\text{swap}(t, i, j) & \\
& 0\{\text{int } a; 1\ a := t[i]; 2\ \text{skip;}
& 3\{\text{int } b; 4\ b := t[j]; 5\ \text{skip;}
& 6\{\text{int } t'; 7\ t' := t[i ↦ b]; 8\ \text{skip;}
& 9\ \text{result := } t'[j ↦ a]^{10}\}^{11}\}^{12}\}^{13}
& \rightarrow (\text{pop}^4(\square), \text{let } b = 1 \text{ in}
& \text{let } t' = t[i ↦ b] \text{ in } t'[j ↦ a])
\end{align*}

\begin{align*}
\text{main}() & \\
& 0\{\text{int } z; 1\ z := +(y, 1); 2\ \text{skip;}
& 3\ \text{result := swap}(x, y, z)^4\}^5
& \rightarrow \text{pop}(\square)
\end{align*}

Stack

\begin{align*}
(b & \mapsto 1, a \mapsto 0, \quad j \mapsto 1, i \mapsto 0, t \mapsto r, \\
\text{result } \mapsto \text{undef}, \quad \parallel \quad z \mapsto 1, y \mapsto 0, x \mapsto r, \\
\text{result } \mapsto \text{undef}, \quad \parallel \quad \text{result } \mapsto \text{undefined})
\end{align*}

Store

\begin{align*}
(r & \mapsto \langle 0, 1 \rangle)
\end{align*}

Count

\begin{align*}
(r & \mapsto 2)
\end{align*}
Expression

\[
\text{pop}^5(
\text{let } b = 1 \text{ in }
\text{let } t' = t[i \mapsto b] \text{ in }
\; t'[j \mapsto a]))
\]
Expression

\[
\text{pop}^5( \\
\text{let } b = 1 \text{ in} \\
\text{let } t' = t[i \mapsto b] \text{ in} \\
t'[j \mapsto a])
\]

Stack

\[
(a \mapsto 0, \\
j \mapsto 1, i \mapsto 0, t \mapsto r, \\
z \mapsto 1, y \mapsto 0, x \mapsto r)
\]

Store

\[
(r \mapsto \langle 0, 1 \rangle)
\]

Count

\[
(r \mapsto 2)
\]
A Small Functional Language
Evaluation with Reference Counting
A Small Imperative Language
Formalization in PVS
Proof $FL \rightarrow RL$ translation both with and without typing

Actually a 4\textsuperscript{th} language: $RL$ with several steps at once

$\approx 7k$ lines definitions or theorems, proof files $\approx 310k$ lines long

Code available at
https://github.com/SRI-CSL/PVSCodegen

Done before my internship: most definitions of $FL$, without proofs
Problems caused by PVS

- PVS is slow: proofs take a long time to check
- PVS has bugs: could prove \texttt{false}
- Proofs of TCCs get misplaced after typechecking again

→ but allowed to detect those bugs
The proved theorems

bisimulation_lemma: THEOREM

\[\text{NOT } tS\text{'}state\text{'}error \text{ AND }\]
\[\text{NOT } trS\text{'}state\text{'}error \text{ AND }\]
\[\text{defs_well_typed}(D, tS\text{'}def\_types) \text{ AND }\]
\[\text{state\_matches?}(\text{typed\_to\_topstate}(tS),\]
\[\text{typed\_to\_topstate}(trS)) \text{ IMPLIES }\]
\[\text{state\_matches?}(\text{typed\_reduce}(D)(tS),\]
\[\text{typed\_reduce\_n}(D)(\]
\[\text{top\_releases\_ct}(trS\text{'}state\text{'}redex) + 1, trS))\]
bisimulation_lemma_i: THEOREM
  NOT trS1\textquoteleft state\textquoteleft error AND NOT trS2\textquoteleft state\textquoteleft error AND
defs_well_typed(D, trS1\textquoteleft def_types) AND
iastate_matches(trS1, trS2) IMPLIES
  EXISTS (n: posnat):
    iastate_matches(typed_reduce_n(D)(n, trS1),
    typed_iareduce(D)(trS2))
The proved theorems

bisimulation_lemma: LEMMA

FORALL (D, (trS |
    defs_well_typed(D, trS`def_types)), iS):
    NOT iS`error AND
    state_matches(D, trS, iS) IMPLIES
    (state_matches(D, trS, reduce(iS)) AND
     max_inst_steps(reduce(iS)) < max_inst_steps(iS))
OR
    state_matches(D, typed_iareduce(D)(trS),
                  reduce(iS))
Conclusion

- Verified code generation from a functional language to an imperative one
- Includes garbage collection and destructive updates
- Article to be submitted to CPP 2018
- Future work: Verify IL to C translation, add first-class functions and closures, try to verify completely PVS2C