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 $\rightarrow$ 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

$$\texttt{let } a = t[i] \texttt{ in let } b = t[j] \texttt{ in let } t' = t[i \mapsto b] \texttt{ in } t'[j \mapsto a]$$

$$\begin{aligned}
e ::= {} & \mid n \\
& \mid x \\
& \mid \texttt{nil} \\
& \mid f(x_1, \ldots, x_n) \\
& \mid \texttt{let } (x : \texttt{int}\star^n) = e_1 \texttt{ in } e_2 \\
& \mid \texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid x[y] \\
& \mid x[y \mapsto z] \\
& \mid \texttt{newint}(n) \mid \texttt{newref}(n) \\
& \mid \texttt{pop}(e_1) \mid \texttt{ref}(k)
\end{aligned}$$

$$\begin{aligned}
K ::= {} & \mid \Box \mid \texttt{pop}(K_1) \\
& \mid \texttt{let } (x : \texttt{int}\star^n) = K_1 \texttt{ in } e_1 \\
v ::= {} & \mid n \mid \texttt{nil} \mid \texttt{ref}(k) \\
r ::= {} & \mid x \\
& \mid f(x_1, \ldots, x_n) \\
& \mid x[y] \\
& \mid x[y \mapsto z] \\
& \mid \texttt{newint}(n) \\
& \mid \texttt{newref}(n) \\
& \mid \texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid \texttt{let } x = v \texttt{ in } e \\
& \mid \texttt{pop}(v)
\end{aligned}$$

Decomposition theorem

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

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

$(e, S, \mathcal{M})$

- e an expression,
- S the stack: maps variables $\rightarrow$ values,
- $\mathcal{M}$ the store: maps references $\rightarrow$ arrays of values.

Reduction of *FL*

- Defined on redexes
- Context-preserving: if $(e, S, \mathcal{M}) \to (e', S', \mathcal{M}')$, then $(Ke, S, \mathcal{M}) \to (Ke', S', \mathcal{M}')$
- Deterministic

Exemples:

$$(x, S, \mathcal{M}) \to (S(x), S, \mathcal{M})$$

$$(x[y \mapsto z], S, \mathcal{M}) \to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]])$$

$$(\texttt{let } x = v \texttt{ in } e, S, \mathcal{M}) \to (\texttt{pop}(e), \mathbf{push}(x, v, S), \mathcal{M})$$

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:

$$\mathcal{C}(\texttt{ref}(k)) = \mathbb{1}_{\texttt{ref}(k) \in e} + \#(S, \texttt{ref}(k)) + \sum_{\texttt{ref}(s) \in \mathcal{M}} \#(\mathcal{M}(\texttt{ref}(s)), \texttt{ref}(k))$$

- 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: $\mathbf{mark}(X, e)$ marks each variable in e that is not in X

$$\mathbf{mark}(X, x) = \begin{cases} x & \text{if } x \in X \\ \underline{x} & \text{otherwise} \end{cases}$$

$$\mathbf{mark}(X, \texttt{let } x = e_1 \texttt{ in } e_2) = \begin{cases} \texttt{let } x = \mathbf{mark}(X \cup \mathbf{vars}(e_2), e_1) \\ \quad \texttt{in } \mathbf{mark}(X, e_2) & \text{if } x \in \mathbf{vars}(e_2) \\ \texttt{let } x = \mathbf{mark}(X \cup \mathbf{vars}(e_2), e_1) \\ \quad \texttt{in release}(\underline{x}, \mathbf{mark}(X, e_2)) & \text{otherwise} \end{cases}$$

- Example:

$$\texttt{let } x = f(y) \texttt{ in ifnz } \underline{z} \texttt{ then } g(\underline{x}, \underline{y}) \texttt{ else release}(\underline{y}, f(\underline{x}))$$

Correctness

Invariants preserved, with states $(e, S, \mathcal{M})$ and $(e', S', \mathcal{M}', \mathcal{C}')$:

- 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 $\texttt{release}(x, e_2)$ of e' have x marked
- There is a function f that maps the references in $\mathcal{M}'$ with count > 0 to references in $\mathcal{M}$ so that:
  - e is obtained by removing `release` and unmarking variables in $f(e')$,
  - For each variable x live in e', $S(x) = f(S'(x))$,
  - For each reference s in $\mathcal{M}'$ with $\mathcal{C}'(s) > 0$, $\mathcal{M}(f(s)) = f(\mathcal{M}'(s))$

A Small Functional Language

Evaluation with Reference Counting

A Small Imperative Language

Formalization in PVS

Syntax

$$\{\texttt{int } z; \{z := x[\underline{y}]; \texttt{result} := \underline{x}[\underline{z}]\}\}$$

$$
\begin{aligned}
e ::= {} & \mid n \\
& \mid x \\
& \mid \texttt{nil} \\
& \mid f(x_1, \ldots, x_n) \\
& \mid x[y] \\
& \mid x[y \mapsto z] \\
& \mid \texttt{newint}(n) \\
& \mid \texttt{newref}(n)
\end{aligned}
$$

$$
\begin{aligned}
s ::= {} & \mid x := e \\
& \mid \texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2 \\
& \mid \texttt{skip} \\
& \mid \{s_1; s_2\} \\
& \mid \texttt{release } x \\
& \mid \{\texttt{int}\star^n\ x; s\}
\end{aligned}
$$

$$\text{decl} ::= \texttt{int}\star^n\ x$$

$$\text{function} ::= (\text{name}, \text{decl}^*, s)$$

$$\text{program} ::= \text{function}^*$$

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 $(\text{function}, \text{program counter}, \text{local depth})$

Reduction of *IL*

- Extract statement at current program counter
- Reduce statement
- For assignments, except calls: like *RL* but store result

IL code generation

- Return variable: $\mathbf{translate}(e, x)$ sets x to result of e

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

$$\mathbf{translate}(\texttt{ifnz}\ y\ \texttt{then}\ e_1\ \texttt{else}\ e_2, x) =$$
$$\texttt{ifnz}\ y\ \texttt{then}\ \mathbf{translate}(e_1, x)\ \texttt{else}\ \mathbf{translate}(e_2, x)$$

$$\mathbf{translate}(\texttt{let}\ (y : \texttt{int}\star^n) = e_1\ \texttt{in}\ e_2, x) =$$
$$\{\texttt{int}\star^n\ y; \{\mathbf{translate}(e_1, y); \{\texttt{skip}; \mathbf{translate}(e_2, x)\}\}\}$$

Correctness

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

Program

$$\texttt{swap}(t, i, j)$$
$$\quad {}^{0}\{\texttt{int } a;^{1}\, a := t[i];^{2}\, \texttt{skip};$$
$$\quad {}^{3}\{\texttt{int } b;^{4}\, b := t[j];^{5}\, \texttt{skip};$$
$$\quad {}^{6}\{\texttt{int}\star\ t';^{7}\, t' := \underline{t}[\underline{i} \mapsto \underline{b}];^{8}\, \texttt{skip};$$
$$\quad {}^{9}\texttt{result} := \underline{t'}[\underline{j} \mapsto \underline{a}]^{10}\}^{11}\}^{12}\}^{13}$$
$$\to (\texttt{pop}^{4}(\square), \texttt{let } b = 1 \texttt{ in}$$
$$\quad \texttt{let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in } \underline{t'}[\underline{j} \mapsto \underline{a}]))$$

$$\texttt{main}()$$
$$\quad {}^{0}\{\texttt{int } z;^{1}\, z := +(y, 1);^{2}\, \texttt{skip};$$
$$\quad {}^{3}\texttt{result} := \texttt{swap}(x, \underline{y}, \underline{z})^{4}\}^{5}$$
$$\to \texttt{pop}(\square)$$

Stack

$$(b \mapsto 1, a \mapsto 0,$$
$$j \mapsto 1, i \mapsto 0, t \mapsto r,$$
$$\texttt{result} \mapsto \texttt{undef}, \|$$
$$z \mapsto 1, y \mapsto 0, x \mapsto r,$$
$$\texttt{result} \mapsto \texttt{undef}, \|$$
$$\texttt{result} \mapsto \texttt{undef})$$

Store

$$(r \mapsto \langle 0, 1 \rangle)$$

Count

$$(r \mapsto 2)$$

Expression

$$\begin{aligned}&\texttt{pop}^5(\\&\texttt{let } b = 1 \texttt{ in}\\&\texttt{let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in}\\&\underline{t'}[\underline{j} \mapsto \underline{a}])\end{aligned}$$

Stack

$$\begin{aligned}&(b \mapsto 1, a \mapsto 0,\\&j \mapsto 1, i \mapsto 0, t \mapsto r,\\&\texttt{result} \mapsto \texttt{undef}, \|\\&z \mapsto 1, y \mapsto 0, x \mapsto r,\\&\texttt{result} \mapsto \texttt{undef}, \|\\&\texttt{result} \mapsto \texttt{undef})\end{aligned}$$

Store

$$(r \mapsto \langle 0, 1 \rangle)$$

Count

$$(r \mapsto 2)$$

Stack

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

Expression

$$\begin{aligned}&\texttt{pop}^5(\\&\texttt{let } b = 1 \texttt{ in}\\&\texttt{let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in}\\&\underline{t'}[\underline{j} \mapsto \underline{a}])\end{aligned}$$

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

PVS formalization

- Proof *FL* $\rightarrow$ *RL* translation both with and without typing
- Actually a 4^{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 `false`
- Proofs of TCCs get misplaced after typechecking again

$\rightarrow$ but allowed to detect those bugs

The proved theorems

```
bisimulation_lemma: THEOREM
  NOT tS`state`error AND
  NOT trS`state`error AND
    defs_well_typed(D, tS`def_types) AND
    state_matches?(typed_to_topstate(tS),
        typed_to_topstate(trS)) IMPLIES
      state_matches?(typed_reduce(D)(tS),
        typed_reduce_n(D)(
          top_releases_ct(trS`state`redex) + 1, trS))
```

The proved theorems

```
bisimulation_lemma_i: THEOREM
  NOT trS1`state`error AND NOT trS2`state`error AND
  defs_well_typed(D, trS1`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