The Correctness of a Code Generator for a Functional Language

Nathanaël Courant, supervised by Natarajan Shankar

March-July 2017

Abstract

This report describes the results of my internship at SRI International, under the supervision of Natarajan Shankar.

The goal of that internship was to formally verify an idealized version of the PVS2C code generator, using PVS. This included defining the semantics of the languages that were to be used, writing the code generator itself, and proving its correctness. The idealized version compiles a first-order, call-by-value functional language to an imperative language with primitives for reference counting.

In the following, we will successively present the different languages used throughout the proof, with their formal semantics, how to compile from one language to the next, and the reasons why they are in bisimulation. Then, we will highlight some parts of the PVS code that were written to prove the theorems mentioned.

Contents

1 Introduction **2**

2 A Small Functional Language **3**

3 Evaluation with Reference Counting **4**

4 A Small Imperative Language **7**

5 Formalization in PVS **11**

6 Conclusions **16**

1 Introduction

Programs written in functional languages are easier to reason with than the corresponding imperative programs, thanks to referential transparency and to the purity of functions. To efficiently implement a functional language on a real-world machine, the most used way is to compile programs from this functional language to programs in an imperative language. However, there are a couple challenges when compiling: indeed, as the evaluation needs to be pure, array updates cannot always be done destructively, which would be unsound, and must sometimes be done by copying the original array and updating the copy. Moreover, memory that is no longer used must be freed to avoid cluttering the memory. Our objective is to prove the correctness of such a transformation, that translates programs from a simple functional programming language *FL* to an imperative language *IL* with primitives for reference-counting.

The functional language *FL* is a first-order language, where all programs are written in A-normal form [3]. It is simply typed, and is also a subset of the intermediate language used in the PVS2C code generator. Its operational semantics is defined as reductions of terms within an evaluation context [1]. This semantics is pure, and each time an array update is made, it copies the array and performs the update on the copy. The translation is done is two steps, one which is a simple static analysis of the program to get a version in *RL*, an annotated form of *FL*, that uses reference counting and makes destructive updates whenever possible. Once this transformation is done, we make a second transformation to get a program in *IL*, which uses assignments and a program counter, but behaves exactly as *RL* in terms of memory.

Although these proofs are operating on idealized versions of languages, their objective is to demonstrate how they can be done, and how we can create similar proofs in other, more complex languages. The proofs have moreover been designed so that more features can be added to the languages used with minimal impact on the proofs.

We give below a brief overview of languages and of the code generator.

Consider the program *swap* which swaps two elements of an array.

$$\begin{aligned}&\texttt{swap}(t, i, j)\\ &\quad = \quad \texttt{let } a = t[i] \texttt{ in let } b = t[j] \texttt{ in let } t' = t[i \mapsto b] \texttt{ in } t'[j \mapsto a]\end{aligned}$$

Although this definition could be simplified in most programming languages, it is not possible here, as we ask that the program is in A-normal form. That is, each subexpression is bound to a variable, so that each expression has only subexpressions that are variables. One of the main advantages of doing so is that is clears any ambiguity on the evaluation order. Evaluation an expression e is done with respect to a stack S, binding variables to values, and a store $\mathcal{M}$, which maps references to arrays. For instance, if e is $\texttt{let } x = y[i \mapsto k] \texttt{ in } x[i]$, and the stack S is $(y \mapsto r, i \mapsto 2, k \mapsto 5)$ with $\mathcal{M} = (r \mapsto \langle 1, 2, 3\rangle)$, e can be decomposed as a redex $y[i \mapsto k]$ in a context $\texttt{let } x = \square \texttt{ in } x[i]$.

The semantics is defined as a set of rewrite rules on redexes, and the property that it is context-preserving. Thus, since we would have $y[i \mapsto k] \to r'$ with the store becoming $(r \mapsto \langle 1, 2, 3\rangle, r' \mapsto \langle 1, 2, 5\rangle)$, we have $e \to \texttt{let } x = r' \texttt{ in } x[i]$ with the same new store. In our case, subsequent reductions would first reduce the `let` expression so that e would become $\texttt{pop}(x[i])$ and S would be $(x \mapsto r', y \mapsto r, i \mapsto 2, k \mapsto 5)$. Here, `pop` is used to keep track of the scope of the variable binding and removing the binding once the variable that is being bound is no longer in scope. The next reductions would reduce e to $\texttt{pop}(5)$, followed by 5 while the binding of x is popped off the stack.

The reference-counting language *RL* behaves likewise, but additionally keeps track of a reference count for each possible reference to the store. In order to do that, it marks the last live occurence of each variable, so that it can free the binding and decrease the reference count at that point. For instance, our previous expression would become $\texttt{let } x = \underline{y}[i \mapsto \underline{k}] \texttt{ in } \underline{x}[\underline{i}]$. Moreover, doing so facilitates destructive updates when updating a reference whose reference count is only 1, and it is the last occurence of the variable being used. Proving that the reference count is accurate is part of the proof of the bisimulation between *FL* and *RL*.

Finally, the last step transforms a program of *RL* into a program of *IL*, an imperative language. It has, by design, primitives quite similar to *RL* itself, but instead of working with context-preserving reductions, it uses a fixed program, assignments, and a program counter. The transformation itself is made in the context of a result variable, say, `result`. In the case of the expression we had previously, it would be translated to $\{\texttt{int}\star x; x := \underline{y}[i \mapsto \underline{k}]; \texttt{skip}; \texttt{result} := \underline{x}[\underline{i}]\}$. The `skip` operation introduced is only there for bookkeeping purposes to help with the proof of the bisimulation. The store and the reference counts behave as in *RL*, but the stack contains additional `result` variables, which hold the results of functions before the end of their execution, as there is no `return` construct but the return is implicit at the end of the function instead. Moreover, the execution uses a fixed program and only keeps a call stack, which is a stack of triples of function identifiers, position of the program counter within the given function, and local stack depth to that function.

Our objective is to construct a simple formalization of the correctness of the code generator. Thus, the languages and the code generator presented here are an idealization of the PVS2C code generator that exists

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

Figure 1: Syntax of *FL*

$$
\begin{aligned}
\text{redex} ::= & \mid x \\
& \mid f(x_1, \ldots, x_n) \\
& \mid x[y] \\
& \mid x[y \mapsto z] \\
& \mid \texttt{newint}(n) \\
& \mid \texttt{newref}(n) \\
& \mid \texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid \texttt{let } x = v \texttt{ in } e \\
& \mid \texttt{pop}(v) \\
& \text{, where } v \text{ is a value.}
\end{aligned}
$$

Figure 2: Definition of a redex

and generates C code from the executable fragment of PVS. The intermediate representations have been chosen to simplify the proofs, while allowing the addition of new constructs without too much overhead.

2 A Small Functional Language

The source functional language *FL* features recursive functions, let-bindings and immutable arrays, and is in A-normal form. Internally we always use de Bruijn indices everywhere for the variables for the simplicity that goes with them; however, in the paper we will use identifiers when giving examples to make the example more readable. Its syntax is defined in Figure 1.

Here, $\texttt{int}\star^n$ means $\texttt{int}\overbrace{\star \ldots \star}^{n}$. For the sake of simplicity, we will often omit the type annotations on the `let` constructors. The `pop` and `ref` constructors are not allowed in programs and are used only during reduction.

We also define a context as the following:

$$K ::= \square \mid \texttt{let } (x : \texttt{int}\star^n) = K_1 \texttt{ in } e_1 \mid \texttt{pop}(K_1)$$

The composition of a context and an expression consists in replacing the hole $\square$ by the expression.

A *value* is a reference, a constant or `nil`. A *redex* is defined in Figure 2.

Theorem 1. *Every expression that is not a value can then be decomposed as the composition of a context and a redex.*

$$
\begin{aligned}
(x, S, \mathcal{M}) &\to (S(x), S, \mathcal{M}) \\
(x[y], S, \mathcal{M}) &\to (\mathcal{M}(S(x))(S(y)), S, \mathcal{M}) \\
(x[y \mapsto z], S, \mathcal{M}) &\to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]]) \\
(\texttt{newint}(n), S, \mathcal{M}) &\to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \langle 0, \dots, 0 \rangle]) \\
(\texttt{newref}(n), S, \mathcal{M}) &\to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \langle \texttt{nil}, \dots, \texttt{nil} \rangle]) \\
(\texttt{let } x = v \texttt{ in } e, S, \mathcal{M}) &\to (\texttt{pop}(e), \mathbf{push}(x, v, S), \mathcal{M}) \\
(\texttt{pop}(v), S, \mathcal{M}) &\to (v, \mathbf{pop}(S), \mathcal{M}) \\
(\texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2, S, \mathcal{M}) &\to \left(\begin{cases} e_1 & \text{if } S(x) \neq 0 \\ e_2 & \text{else} \end{cases}, S, \mathcal{M} \right) \\
(f(x_1, \dots, x_n), S, \mathcal{M}) &\to (v, S, \mathcal{M}) \text{ for primitive } f, \text{ where } f(S(x_1), \dots, S(x_n)) \xrightarrow{\delta} v \\
(f(x_1, \dots, x_n), S, \mathcal{M}) &\to (\texttt{pop}^n(e), \mathbf{push}(a_n, S(x_n), \dots, \mathbf{push}(a_1, S(x_1), S) \dots), \mathcal{M}) \\
&\qquad \text{where } e \text{ is the body of } f,\ a_1, \dots, a_n \text{ its arguments}
\end{aligned}
$$

Figure 3: Semantics of *FL*

The state is defined as a triplet $(e, S, \mathcal{M})$, where e is an expression, S is the stack, which maps variables to values, and $\mathcal{M}$ the state of the memory, which maps a finite number of references to finite sequences of values.

We denote by $\mathbf{new}(\mathcal{M})$ a reference that is not yet defined in $\mathcal{M}$, and $S(x)$ as the element on the stack corresponding to x.We use $\xrightarrow{\delta}$ for a reduction relation that describes how primitive functions such as + work.

The small-step semantics are defined as the unique context-preserving relation $\to$ that is defined on redexes as in Figure 3. It is easily seen that it is deterministic.

It is an error to access or modify outside the bounds given by the store, as well as to call a non-existent function or to call them with an incorrect number of arguments, or to use primitive operations with unsupported arguments. The state obtained after such erroneous reductions will be denoted as $\bot$.

For instance, suppose $\texttt{swap}(t, i, j)$ is $\texttt{let } a = t[i] \texttt{ in let } b = t[j] \texttt{ in let } t' = t[i \mapsto b] \texttt{ in } t'[j \mapsto a]$. Suppose that $e = \texttt{let } z = +(y, 1) \texttt{ in swap}(x, y, z)$ with $S = (y \mapsto 0, x \mapsto r)$ and $\mathcal{M} = (r \mapsto \langle 0, 1 \rangle)$. Steps of the reduction are detailed in Figure 4.

3 Evaluation with Reference Counting

For the reference-counting language *RL*, we now have an additional constructor, $\texttt{release}(x, B)$, which is a redex as well. In addition, each variable can be *marked*, meaning this occurence of the variable is the last. We maintain as an invariant the count, written $\mathcal{C}$, for each reference. It is defined as follows:

Defining $\#(S, x)$ as the number of times x appears in the sequence S, we have:

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

The advantage of defining and maintaining this reference count is to be able to free memory once it is no longer needed, and to be able to perform more efficient destructive updates on the arrays.

The evaluation also preserves those three invariants:

early-release. Each variable in S that is no longer live in e is not bound to a reference.

correct-marking. The expression e is correctly marked (deleting all the markings in e and marking it again returns an expression identical to e).

release-marked. All subterms of e of the form $\texttt{release}(x, B)$ have that occurrence of x marked[1].

[1]Together with the **correct-marking** invariant, this implies that x does not occur as a free variable in B.

$$
\begin{aligned}
& (\texttt{let } z = +(y,1) \texttt{ in } \texttt{swap}(x,y,z), (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\texttt{let } z = 1 \texttt{ in } \texttt{swap}(x,y,z), (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\texttt{pop}(\texttt{swap}(x,y,z)), (z \mapsto 1, y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } a = t[i] \texttt{ in } \dots, (j \mapsto 1, i \mapsto 0, t \mapsto r, \dots), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } a = 0 \texttt{ in } \dots, (j \mapsto 1, i \mapsto 0, t \mapsto r, \dots), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } b = t[j] \texttt{ in } \dots, (a \mapsto 0, j \mapsto 1, \dots), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } b = 1 \texttt{ in } \dots, (a \mapsto 0, j \mapsto 1, \dots), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } t' = t[i \mapsto b] \texttt{ in } \dots, (b \mapsto 1, a \mapsto 0, \dots), (r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots \texttt{let } t' = r' \texttt{ in } \dots, (b \mapsto 1, a \mapsto 0, \dots), (r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle)) \\
\longrightarrow\ & (\dots t'[j \mapsto a] \dots, (t' \mapsto r', b \mapsto 1, \dots), (r' \mapsto \langle 1,1\rangle, \dots)) \\
\longrightarrow\ & (\dots r'' \dots, (t' \mapsto r', b \mapsto 1, \dots), (r'' \mapsto \langle 1,0\rangle, \dots)) \\
\longrightarrow^{+}\ & (r'', (y \mapsto 0, x \mapsto r), (r'' \mapsto \langle 1,0\rangle, r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle))
\end{aligned}
$$

Figure 4: An example reduction

We use the following helper functions:

$$
\begin{aligned}
\mathbf{incr}(v,\mathcal{C}) &= \mathcal{C}[v \mapsto \mathcal{C}(v)+1] && \text{if } v \text{ is a reference} \\
\mathbf{incr}(v,\mathcal{C}) &= \mathcal{C} && \text{otherwise} \\
\mathbf{decr}(v,\mathcal{C}) &= \mathcal{C}[v \mapsto \mathcal{C}(v)-1] && \text{if } v \text{ is a reference} \\
\mathbf{decr}(v,\mathcal{C}) &= \mathcal{C} && \text{otherwise}
\end{aligned}
$$

The function **decr_rec** takes a value, the state of memory and a count, and if the value is a reference, decreases its count. In case the count is zero, it recursively decreases the count of all the references pointed by that one and replaces them by `nil`.

A few instances of reductions updated to preserve the invariants are presented in Figure 5.

The reduction of *update* redexes is by far the most complicated of all; but it is also the main reason why we perform this step of reference counting.

For instance, suppose $\texttt{swap}(t,i,j)$ is $\texttt{let } a = t[i] \texttt{ in let } b = t[j] \texttt{ in let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in } \underline{t'}[\underline{j} \mapsto \underline{a}]$. Suppose that $e = \texttt{let } z = +(y,1) \texttt{ in } \texttt{swap}(\underline{x},\underline{y},\underline{z})$ with $S = (y \mapsto 0, x \mapsto r)$, $\mathcal{M} = (r \mapsto \langle 0,1\rangle)$ and $\mathcal{C} = (r \mapsto 2)$. Steps of the reduction are detailed in Figure 6.

Notice how even though the reference count of r was 2 initially, we still saved a copy compared to Figure 4 and made a destructive update instead. Indeed, the reference count of the result of an array update is *always* 1. Either it is the result of a destructive update, in which case the reference count has to be 1, or it is a fresh copy, in which case the count is 1 as well.

Theorem 2. *With the reductions in Figure 5, it is an invariant that the count is accurate, that is, equation 1 holds, and the other invariants are preserved as well. It is also possible to adapt the reductions to preserve the same invariants.*

To translate an expression from the original version to the one with reference counting, we mark the last occurence of each variable on each execution path, inserting `release` constructors if needed for `ifnz` branches. This also corresponds to marking the last live occurrence of each variable.

$$
\begin{aligned}
(x, S, \mathcal{M}, \mathcal{C}) &\to (S(x), S, \mathcal{M}, \mathbf{incr}(S(x), \mathcal{C})) \\
&\qquad\qquad \text{if } x \text{ is not marked and } S(x) \text{ is a reference} \\
(x, S, \mathcal{M}, \mathcal{C}) &\to (S(x), S[x \mapsto \texttt{nil}], \mathcal{M}, \mathcal{C}) \qquad \text{if } x \text{ is marked and } S(x) \text{ is a reference} \\
(x, S, \mathcal{M}, \mathcal{C}) &\to (S(x), S, \mathcal{M}, \mathcal{C}) \qquad \text{otherwise} \\
(x[y \mapsto z], S, \mathcal{M}, \mathcal{C}) &\to (S(x), S'[x \mapsto \texttt{nil}], \mathbf{decr\_rec}(\mathcal{M}(S(x))(S(y)), \\
&\qquad\qquad (\mathcal{M}[S(x) \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]], \\
&\qquad\qquad \mathbf{incr}(S(z), \mathcal{C}')))) \qquad \text{if } \mathcal{C}(x) = 1 \text{ and } S(x) \neq S(z) \text{ and } x \text{ is marked} \\
(x[y \mapsto z], S, \mathcal{M}, \mathcal{C}) &\to (\mathbf{new}(\mathcal{M}), S'', \\
&\qquad\qquad \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]], \\
&\qquad\qquad \mathbf{decr}(\mathcal{M}(S(x))(S(y)), \mathbf{incr}(S(z), \\
&\qquad\qquad\quad (\#(\mathcal{M}(S(x)), \cdot) + \mathcal{C}'')[\mathbf{new}(\mathcal{M}) \mapsto 1]))) \qquad \text{otherwise} \\
&\qquad \text{where } (\mathcal{C}', S') = (\mathbf{decr}(S(z), \mathcal{C}), S[z \mapsto \texttt{nil}]) \\
&\qquad\qquad\qquad\qquad \text{if } z \text{ is marked and } S(z) \text{ is a reference} \\
&\qquad\qquad (\mathcal{C}', S') = (\mathcal{C}, S) \qquad \text{otherwise} \\
&\qquad \text{where } (\mathcal{C}'', S'') = (\mathbf{decr}(S(x), \mathcal{C}', S[x \mapsto \texttt{nil}])) \qquad \text{if } x \text{ is marked} \\
&\qquad\qquad (\mathcal{C}'', S'') = (\mathcal{C}', S') \qquad \text{otherwise} \\
(\texttt{release}(\underline{x}, e), S, \mathcal{M}, \mathcal{C}) &\to (e, S[x \mapsto \texttt{nil}], \mathbf{decr\_rec}(S(x), (\mathcal{M}, \mathcal{C}))) \qquad \text{if } S(x) \text{ is a reference} \\
(\texttt{release}(\underline{x}, e), S, \mathcal{M}, \mathcal{C}) &\to (e, S, \mathcal{M}, \mathcal{C}) \qquad \text{otherwise}
\end{aligned}
$$

Figure 5: Semantics of some reductions in RL

$$
\begin{aligned}
& (\texttt{let } z = +(y, 1) \texttt{ in } \texttt{swap}(\underline{x}, \underline{y}, \underline{z}), (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\texttt{let } z = 1 \texttt{ in } \texttt{swap}(\underline{x}, \underline{y}, \underline{z}), (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\texttt{pop}(\texttt{swap}(\underline{x}, \underline{y}, \underline{z})), (z \mapsto 1, y \mapsto 0, x \mapsto r), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } a = t[i] \texttt{ in } \ldots, (j \mapsto 1, i \mapsto 0, t \mapsto r, \ldots, x \mapsto \texttt{nil}), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } a = 0 \texttt{ in } \ldots, (j \mapsto 1, i \mapsto 0, t \mapsto r, \ldots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } b = t[j] \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } b = 1 \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in } \ldots, (b \mapsto 1, a \mapsto 0, \ldots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow\ & (\ldots \texttt{let } t' = r' \texttt{ in } \ldots, (b \mapsto 1, a \mapsto 0, \ldots, t \mapsto \texttt{nil}, \ldots), (r' \mapsto \langle 1, 1\rangle, r \mapsto \langle 0, 1\rangle), (r' \mapsto 1, r \mapsto 1)) \\
\longrightarrow\ & (\ldots \underline{t'}[\underline{j} \mapsto \underline{a}] \ldots, (t' \mapsto r', b \mapsto 1, \ldots), (r' \mapsto \langle 1, 1\rangle, \ldots)) \\
\longrightarrow\ & (\ldots r' \ldots, (t' \mapsto \texttt{nil}, b \mapsto 1, \ldots), (r' \mapsto \langle 1, 0\rangle, \ldots), (r' \mapsto 1, \ldots)) \\
\longrightarrow^{+}\ & (r', (y \mapsto 0, x \mapsto \texttt{nil}), (r' \mapsto \langle 1, 0\rangle, r \mapsto \langle 0, 1\rangle), (r' \mapsto 1, r \mapsto 1))
\end{aligned}
$$

Figure 6: An example reduction in RL

$$
\begin{array}{rl}
e ::= & \mid n \\
& \mid x \\
& \mid \texttt{nil} \\
& \mid f(x_1, \ldots, x_n) \\
& \mid x[y] \\
& \mid x[y \mapsto z] \\
& \mid \texttt{newint}(n) \\
& \mid \texttt{newref}(n)
\end{array}
\qquad
\begin{array}{rl}
s ::= & \mid x := e \\
& \mid \texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2 \\
& \mid \texttt{skip} \\
& \mid \{s_1; s_2\} \\
& \mid \texttt{release } x \\
& \mid \{\texttt{int}\star^n\ x; s\}
\end{array}
$$

$$
\begin{array}{rl}
\text{decl} ::= & \texttt{int}\star^n\ x \\
\text{function} ::= & (\text{name}, \text{decl}^*, s) \\
\text{program} ::= & \text{function}^*
\end{array}
$$

Figure 7: Syntax of *IL*

For instance, underlined variables representing marked variables, we have the following translation:

$$
\begin{array}{l}
\texttt{let } x = g(w) \texttt{ in} \\
\texttt{let } t = \\
\quad \texttt{ifnz } x \texttt{ then} \\
\quad\quad f(y, z) \\
\quad \texttt{else} \\
\quad\quad f(y, s) \\
\texttt{in} \\
f(t, s)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\texttt{let } x = g(\underline{w}) \texttt{ in} \\
\texttt{let } t = \\
\quad \texttt{ifnz } \underline{x} \texttt{ then} \\
\quad\quad f(\underline{y}, \underline{z}) \\
\quad \texttt{else} \\
\quad\quad \texttt{release}(\underline{z}, f(\underline{y}, s)) \\
\texttt{in} \\
f(\underline{t}, \underline{s})
\end{array}
$$

To establish a bisimulation between the reference-counting version $(e', S', \mathcal{M}', \mathcal{C}')$ and the original one $(e, S, \mathcal{M})$, we say these two states match if there exists a translation function f from the elements of the domain of $\mathcal{M}'$ with a count greater than zero to those of the domain of $\mathcal{M}$ such that:

- The expression e is the result of translating the references (applying f to each of the references) when unmarking all the variables and removing all `release` constructors in e',
- For each variable x that is live in e', $S(x)$ is the result of translating the references of $S'(x)$,
- For each reference s in the domain of $\mathcal{M}'$ with a count greater than zero, $\mathcal{M}(f(s))$ is the result of translating the references of $\mathcal{M}'(s)$.

Note that although it is not required for the translation function to be injective, it is actually an invariant (that we however do not need to prove to obtain the bisimulation result).

Theorem 3. *If the state $\mathcal{S} = (e, S, \mathcal{M})$ matches the state $\mathcal{S}' = (e', S', \mathcal{M}', \mathcal{C}')$, we have:*

- *if the current redex of e' is a `release` redex, then the state obtained when reducing $\mathcal{S}'$ after one step still matches the state $\mathcal{S}$,*
- *if it is not a `release` redex, then the state obtained when reducing $\mathcal{S}$ and the one obtained when reducing $\mathcal{S}'$ for a step each still match each other.*

Theorem 4. *The reduction relations $\longrightarrow$ in* FL *and $\longrightarrow^+$ in* RL *are in bisimulation.*

4 A Small Imperative Language

The target of our code generation will be an imperative language *IL*, with some high-level primitives that keep track of reference counts. From there, we can go to a lower-level imperative language such as C that does not keep track of reference counts automatically. A program *IL* is defined as a sequence of functions, whose body is a statement, with the definitions in Figure 7.

As in *RL*, variables can be marked. A *value* is now either `nil`, an integer, a reference or `undef`. It is an error to use the value `undef` in any expression.

Once a program with definitions Δ is fixed, the state is now a quadruplet $(\mathcal{R}, S, \mathcal{M}, \mathcal{C})$, where as previously, S is the stack, which maps variables to values, $\mathcal{M}$ and $\mathcal{C}$ are the store and the reference counts, respectively. $\mathcal{R}$ is the *call stack*, which holds triplets composed of a function identifier, the position of the program counter within that function, and the local stack depth in that function.

$$
\begin{aligned}
\mathbf{length}(\{s_1; s_2\}) &= \mathbf{length}(s_1) + \mathbf{length}(s_2) \\
\mathbf{length}(\{\texttt{int}\star^n x; s\}) &= 2 + \mathbf{length}(s) \\
\mathbf{length}(\texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2) &= 1 + \mathbf{length}(s_1) + \mathbf{length}(s_2) \\
\mathbf{length}(s) &= 1 \qquad \text{otherwise}
\end{aligned}
$$

Figure 8: Definition of **length**

$$
\begin{aligned}
\mathbf{extract}(\{s_1; s_2\}, pc) &= \begin{cases} \mathbf{extract}(s_1, pc) & \text{if } pc < \mathbf{length}(s_1) \\ \mathbf{extract}(s_2, pc - \mathbf{length}(s_1)) & \text{otherwise} \end{cases} \\
\mathbf{extract}(\{\texttt{int}\star^n x; s\}, pc) &= \begin{cases} \mathbf{extract}(s, pc - 1) & \text{if } 0 < pc < 1 + \mathbf{length}(s) \\ (\{\texttt{int}\star^n x; s\}, pc) & \text{otherwise} \end{cases} \\
\mathbf{extract}(\texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2, pc) &= \begin{cases} (\texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2, 0) & \text{if } pc = 0 \\ \mathbf{extract}(s_1, pc - 1) & \text{if } 0 < pc < 1 + \mathbf{length}(s_1) \\ \mathbf{extract}(s_2, pc - 1 - \mathbf{length}(s_1)) & \text{otherwise} \end{cases} \\
\mathbf{extract}(s, pc) &= (s, 0) \qquad \text{otherwise}
\end{aligned}
$$

Figure 9: Definition of **extract**

We define the **length** of a statement in Figure 8. We also define **extract** and **next** to respectively extract the statement at the current position of the program counter, and to move the program counter to its next position (both are defined for $pc < \mathbf{length}(s)$) in Figures 9 and 10.

Examples of how these functions work are given in Figure 11. Note that if $(t, j) = \mathbf{extract}(s, pc)$ and $j \neq 0$, then t is a statement of the form $\{\texttt{int}\star^n\ x; t'\}$ and $j = \mathbf{length}(t) - 1 = \mathbf{length}(t') + 1$.

The semantics of *IL* is defined in Figure 12. Notice that there is a variable pushed implicitly on the stack before the arguments when calling a function, which is used to hold the return value of the function, as there is no explicit `return` operation. In the following, we shall call that variable `result` whenever we need a name for it.

To translate a function with body e from *RL* to the *IL*, we use $\mathbf{translate}(e, \texttt{result})$, where **translate** is defined in Figure 13. Note that in the case of the translation of `let` constructs, the variable bound by the `let` already exists in the branch of the `let` where we need to define it, so that the de Bruijn indices in *RL* and *IL* are not the same. The `skip` introduced in the `let` translation is there to make the proof of bisimulation easier.

Taking once again the example of our original *swap* program, once translated, we have the program seen in Figure 14a. The steps of its reduction are detailed in Figure 14b.

Finally, to prove that the imperative program obtained from translation behaves as the original program, we notice the following facts:

- Given the position of the program counter and stack, it is possible to reconstruct the redex and context for a single function,
- Given the position of the program counter inside a function that is calling another, it is possible to reconstruct the context for this function,
- Given these positions, it is also possible to reconstruct a mapping from all the variables defined at that point in *RL* and *IL* such for each of these variables, the value in both stacks are the same (with variable names, the mapping is the identity, but not with de Bruijn indices),
- If we have passed the point where $\mathbf{translate}(A, x)$ sets the variable x, then the variable x should not have the value `undef`
- At all times, the store and counts are the same in both languages.

Showing that all these invariants are preserved makes it possible to prove the following:

Theorem 5. *The reduction relations* $\longrightarrow$ *in* RL *and* $\longrightarrow^+$ *in* IL *are in bisimulation.*

$$\mathbf{next}(\{s_1; s_2\}, pc) = \begin{cases} \mathbf{next}(s_1, pc) & \text{if } pc < \mathbf{length}(s_1) \\ \mathbf{length}(s_1) + \mathbf{next}(s_2, pc - \mathbf{length}(s_1)) & \text{otherwise} \end{cases}$$

$$\mathbf{next}(\{\texttt{int} \star^n x; s\}, pc) = \begin{cases} 1 & \text{if } pc = 0 \\ 1 + \mathbf{next}(s, pc - 1) & \text{if } 0 < pc < 1 + \mathbf{length}(s) \\ 2 + \mathbf{length}(s) & \text{otherwise} \end{cases}$$

$$\mathbf{next}(\texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2, pc) = \begin{cases} \bot & \text{if } pc = 0 \\ 1 + \mathbf{next}(s_1, pc - 1) & \text{if } 0 < pc < 1 + \mathbf{length}(s_1) \\ & \wedge\ \mathbf{next}(s_1, pc - 1) < \mathbf{length}(s_1) \\ 1 + \mathbf{length}(s_1) + \mathbf{length}(s_2) & \text{if } 0 < pc < 1 + \mathbf{length}(s_1) \\ & \wedge\ \mathbf{next}(s_1, pc - 1) = \mathbf{length}(s_1) \\ 1 + \mathbf{next}(s_2, pc - 1 - \mathbf{length}(s_1)) & \text{otherwise} \end{cases}$$

$$\mathbf{next}(s, pc) = 1 \qquad \text{otherwise}$$

Figure 10: Definition of the **next** function.

$^0\{\texttt{int } x;$
$\quad {}^1\texttt{ifnz } y \texttt{ then}$
$\quad\quad {}^2\{\texttt{int } w;^3 w := f(y)^4\}$
$\quad \texttt{else}$
$\quad\quad \{^5 x := t;^6 y := x\}$
$^7\}^8$

(a) The statement s, with annotated positions for the program counter. We have $\mathbf{length}(s) = 8$.

$\mathbf{extract}(s, 1) = (\texttt{ifnz } y \ldots, 0)$	$\mathbf{next}(s, 1) = \bot$
$\mathbf{extract}(s, 2) = (\{\texttt{int } w; \ldots\}, 0)$	$\mathbf{next}(s, 2) = 3$
$\mathbf{extract}(s, 3) = (w := f(y), 0)$	$\mathbf{next}(s, 3) = 4$
$\mathbf{extract}(s, 4) = (\{\texttt{int } w; \ldots\}, 2)$	$\mathbf{next}(s, 4) = 7$
$\mathbf{extract}(s, 6) = (y := x, 0)$	$\mathbf{next}(s, 6) = 7$
$\mathbf{extract}(s, 7) = (s, 7)$	$\mathbf{next}(s, 7) = 8$

(b) Some values of **extract** and **next**.

Figure 11: Examples of the values of **length**, **extract** and **next**

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, pc, depth - 1) :: \mathcal{R}', \mathbf{pop}(S), \mathcal{M}, \mathcal{C}) \quad \text{if } pc = \mathbf{length}(\Delta(f)) \wedge depth > 1$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ([], [\mathbf{top}(S)], \mathcal{M}, \mathcal{C}) \quad \text{if } pc = \mathbf{length}(\Delta(f)) \wedge depth = 1 \wedge \mathcal{R}' = []$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((g, \mathbf{next}(\Delta(g), pc'), depth') :: \mathcal{R}'', \mathbf{pop}(S)[x \mapsto \mathbf{top}(S)], \mathcal{M}, \mathcal{C})$$
$$\text{if } pc = \mathbf{length}(\Delta(f)) \wedge depth = 1 \wedge \mathcal{R}' \neq []$$
$$\text{where } (g, pc', depth') :: \mathcal{R}'' = \mathcal{R}'$$
$$(x := f(x_1, \ldots, x_n), 0) = \mathbf{extract}(\Delta(g), pc')$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth) :: \mathcal{R}', S, \mathcal{M}, \mathcal{C}) \quad \text{if } s = \texttt{skip}$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth + 1) :: \mathcal{R}', \mathbf{push}(x, \texttt{undef}, S), \mathcal{M}, \mathcal{C}) \quad \text{if } s = \{\texttt{int}\star^n x; s\} \wedge j = 0$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth - 1) :: \mathcal{R}', \mathbf{pop}(S), \mathcal{M}, \mathcal{C}) \quad \text{if } s = \{\texttt{int}\star^n x; s\} \wedge j \neq 0$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, pc + 1, depth) :: \mathcal{R}', S, \mathcal{M}, \mathcal{C}) \quad \text{if } s = \texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2 \wedge S(x) \neq 0$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, pc + 1 + \mathbf{length}(s_1), depth) :: \mathcal{R}', S, \mathcal{M}, \mathcal{C})$$
$$\text{if } s = \texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2 \wedge S(x) = 0$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((g, 0, n + 1) :: \mathcal{R},$$
$$\mathbf{push}(a_n, S(x_n), \ldots \mathbf{push}(a_1, S(x_1), \mathbf{push}(\texttt{result}, \texttt{undef}, S') \ldots), \mathcal{M}, \mathcal{C}')$$
$$\text{if } s = x := g(x_1, \ldots, x_n) \text{ and } g \text{ is not a primitive operation}$$

$$\text{where } S'(y) = \begin{cases} \texttt{nil} & \text{if } y \text{ is marked in } x_1, \ldots, x_n \text{ and } S(y) \text{ is a reference} \\ S(y) & \text{otherwise} \end{cases}$$
$$\mathcal{C}'(r) = \mathcal{C}(r) + |\{i | x_i \text{ is not marked } \wedge S(x_i) = r\}|$$
$$(a_1, \ldots, a_n) \text{ are the arguments of } g$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth) :: \mathcal{R}', S'[x \mapsto v], \mathcal{M}', \mathcal{C}') \quad \text{if } s = x := e$$
$$\text{where } (e, S, \mathcal{M}, \mathcal{C}) \to^{\leq 1} (v, S', \mathcal{M}', \mathcal{C}') \text{ in } RL$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth) :: \mathcal{R}', S, \mathcal{M}, \mathcal{C}) \quad \text{if } s = \texttt{release } x \text{ and } S(x) \text{ is not a reference}$$

$$(\mathcal{R}, S, \mathcal{M}, \mathcal{C}) \to ((f, npc, depth) :: \mathcal{R}', S[x \mapsto \texttt{nil}], \mathbf{decr\_rec}((\mathcal{M}, \mathcal{C}), S(x)))$$
$$\text{if } s = \texttt{release } x \text{ and } S(x) \text{ is a reference}$$

where

$$(f, pc, depth) :: \mathcal{R}' = \mathcal{R},$$
$$(s, j) = \mathbf{extract}(\Delta(f), pc),$$
$$npc = \mathbf{next}(\Delta(f), pc)$$

Figure 12: Semantics of *IL*

$$\begin{aligned}
\textbf{translate}(n, x) &= x := n \\
\textbf{translate}(y, x) &= x := y \\
\textbf{translate}(\texttt{nil}, x) &= x := \texttt{nil} \\
\textbf{translate}(f(x_1, \ldots, x_n), x) &= x := f(x_1, \ldots, x_n) \\
\textbf{translate}(\texttt{let } (y : \texttt{int}\star^n) = e_1 \texttt{ in } e_2, x) &= \{\texttt{int}\star^n\ y; \{\textbf{translate}(e_1, y); \{\texttt{skip}; \textbf{translate}(e_2, x)\}\}\} \\
\textbf{translate}(\texttt{ifnz } y \texttt{ then } e_1 \texttt{ else } e_2, x) &= \texttt{ifnz } y \texttt{ then } \textbf{translate}(e_1, x) \texttt{ else } \textbf{translate}(e_2, x) \\
\textbf{translate}(y[z], x) &= x := y[z] \\
\textbf{translate}(y[z \mapsto w], x) &= x := y[z \mapsto w] \\
\textbf{translate}(\texttt{newint}(n), x) &= x := \texttt{newint}(n) \\
\textbf{translate}(\texttt{newref}(n), x) &= x := \texttt{newref}(n) \\
\textbf{translate}(\texttt{release}(y, e), x) &= \{\texttt{release } y; \textbf{translate}(e, x)\}
\end{aligned}$$

Figure 13: Translation from *RL* to *IL*

5 Formalization in PVS

The following datatypes are used for the expressions and contexts of *FL* and *RL* (with the variable marks and the `release` constructors being used only in the reference-counting version).

```
IExpression: DATATYPE
BEGIN
  variable(index: nat, marked: bool): variable?
  constant(value: int): constant?
  nil: nil?
  application(fun: nat, args: list[(variable?)]): application?
  letexpr(letrhs:  IExpression, body: IExpression): letexpr?
  ift(condition: (variable?), thenexpr, elseexpr: IExpression): ift?
  update(target, lhs, rhs: (variable?)): update?
  lookup(arrayvalue, position: (variable?)): lookup?
  newint(size: nat): newint?
  newref(size: nat): newref?
  pop(pbody: IExpression): pop?
  ref(refindex: nat): ref?
  release(rvar: (variable?), rexpr: IExpression): release?
END IExpression

IContext: DATATYPE
BEGIN
  hole: hole?
  letc(letcrhs: IContext, letcbody: IExpression): letc?
  popc(pcbody: IContext): popc?
END IContext
```

Note that the types are not included in these definitions; we actually have both an untyped version, which is the one presented above, and a typed version which is only a typing overlay on top of the untyped version.

The following is the statement of Theorem 1.

```
context_lemma: LEMMA
  value?(A) OR (EXISTS K, B: redex?(B) AND A = fill(K, B))
```

We have a reduction function for each kind of redex; the small-step semantics are then defined as extracting the redex inside the current expression, and reducing it (this can affect other parts of the state as well, such as the stack and the store) while preserving the context. For instance, the definition below shows the reduction of a *variable* redex, as in Figure 3.[2]. The type `goodstate` is defined as the subset of states that are not an error state.

```
variableReduce(D)(gS: goodstate | variable?(gS`redex)): estate =
  gS WITH [`redex := get(gS`stack)(gS`redex)]
```

[2]In PVS, `gS`redex` means the record access to field `redex` of `gS` and `gS WITH [`redex := x]` means a record update.

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

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

(a) Translation of the *swap* program (removing unnecessary braces for clarity). The exponents show the position of the program counter.

$$
\begin{array}{ll}
 & ([(\texttt{main}, 0, 3)], (y \mapsto 0, x \mapsto r, \texttt{result} \mapsto \texttt{undef}, \texttt{result} \mapsto \texttt{undef}), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{main}, 1, 4)], (z \mapsto \texttt{undef}, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{main}, 2, 4)], (z \mapsto 1, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{main}, 3, 4)], (z \mapsto 1, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 0, 4), (\texttt{main}, 3, 4)], (j \mapsto 1, i \mapsto 0, t \mapsto r, \texttt{result} \mapsto \texttt{undef}, \dots, x \mapsto \texttt{nil}, \dots), \\
 & \qquad (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 1, 5), \dots], (a \mapsto \texttt{undef}, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 2, 5), \dots], (a \mapsto 0, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 3, 5), \dots], (a \mapsto 0, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 4, 6), \dots], (b \mapsto \texttt{undef}, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow^{+} & ([(\texttt{swap}, 6, 6), \dots], (b \mapsto 1, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow & ([(\texttt{swap}, 7, 7), \dots], (t' \mapsto \texttt{undef}, \dots), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2)) \\
\longrightarrow^{+} & ([(\texttt{swap}, 9, 7), \dots], (t' \mapsto r', \dots, t \mapsto \texttt{nil}, \dots), (r' \mapsto \langle 1, 1\rangle, r \mapsto \langle 0, 1\rangle), (r' \mapsto 1, r \mapsto 1)) \\
\longrightarrow & ([(\texttt{swap}, 10, 7), \dots], (t' \mapsto \texttt{nil}, \dots, \texttt{result} \mapsto r', \dots), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow^{+} & ([(\texttt{swap}, 13, 4), \dots], (j \mapsto 1, \dots), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow^{+} & ([(\texttt{swap}, 13, 1), \dots], (\texttt{result} \mapsto r', \dots), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow & ([(\texttt{main}, 4, 4)], (z \mapsto 1, \dots, \texttt{result} \mapsto r', \dots), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow & ([(\texttt{main}, 5, 3)], (y \mapsto 0, \dots,), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow^{+} & ([(\texttt{main}, 5, 1)], (\texttt{result} \mapsto r', \texttt{result} \mapsto \texttt{undef}), (r' \mapsto \langle 1, 0\rangle, \dots), (r' \mapsto 1, \dots)) \\
\longrightarrow & ([], (\texttt{result} \mapsto r'), (r' \mapsto \langle 1, 0\rangle, r \mapsto \langle 0, 1\rangle), (r' \mapsto 1, r \mapsto 1))
\end{array}
$$

(b) Reduction of the imperative version of the *swap* program. See Figure 14a for the definitions of the functions and the program counters positions.

Figure 14: Translation of the *swap* program and its reduction

Suppose the state is:

$([(\texttt{swap}, 5, 6), (\texttt{main}, 3, 4)], (b \mapsto 1, a \mapsto 0, j \mapsto 1, i \mapsto 0, t \mapsto r, \texttt{result} \mapsto \texttt{undef},$
$z \mapsto 1, y \mapsto 0, x \mapsto \texttt{nil}, \texttt{result} \mapsto \texttt{undef}, \texttt{result} \mapsto \texttt{undef}), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2))$

We extract the following:

$(\texttt{swap}, 5, 6) \rightsquigarrow (e = \texttt{let } b = 1 \texttt{ in let } t' = \underline{t}[\underline{i} \mapsto \underline{b}] \texttt{ in } \underline{t'}[\underline{j} \mapsto \underline{a}], \texttt{pop}(\texttt{pop}(\texttt{pop}(\texttt{pop}(\Box)))))$

$(\texttt{main}, 3, 4) \rightsquigarrow \texttt{pop}(\Box)$

Thus this state matches the state below we saw in the reduction of the *swap* program in *RL*.

$(\texttt{pop}^5(e), (a \mapsto 0, j \mapsto 1, i \mapsto 0, t \mapsto r, z \mapsto 1, y \mapsto 0, x \mapsto \texttt{nil}), (r \mapsto \langle 0, 1\rangle), (r \mapsto 2))$

Figure 15: Bisimulation for the *swap* program

The top-level reduction function `reduce` performs a case analysis on the type of redex. The function `to_topstate` converts from the decomposition as an expression and a context to a single expression by filling the hole of the context with the expression, while `make_redex_e` does the reverse and creates a redex and a context from its argument.

```
reduce(D)(gS: goodstate): estate =
  LET t = to_topstate(gS) IN
  IF value?(t`redex) THEN
    t
  ELSE
    LET (nS: goodstate) = make_redex_e(t) IN
    IF variable?(nS`redex) THEN
      variableReduce(D)(nS)
    ELSIF pureLetRedex?(nS`redex) THEN
      letReduce(D)(nS)
    ELSIF applyRedex?(nS`redex) THEN
      applyReduce(D)(nS)
    ELSIF [...]
```

The reduction in the reference-counting version is very similar, but the reduction functions are a bit more complicated. For example, the reduction of a *variable* redex is shown below.

```
variableReduce(D)(grS | variable?(grS`redex)): rstate =
  LET stack = grS`stack,
      store = grS`store,
      expr = grS`redex
  IN
  LET value : (domainValue?(grS`domain)) = get(stack)(expr) IN
  IF ref?(value) THEN
    IF marked(grS`redex) THEN
      grS WITH [`redex := value,
                `stack`seq(stack`length - 1 - index(expr)) := nil]
    ELSE
      grS WITH [`redex := value,
                `count(refindex(value)) := grS`count(refindex(value)) + 1]
    ENDIF
  ELSE
    grS WITH [`redex := value]
  ENDIF
```

Theorem 2 is partly enforced (equation 1 and the invariant (correct-marking)) by the types themselves, relying heavily on PVS dependent typing features. The other two invariants are enforced by the following lemmas.

```
reduce_ndr: JUDGEMENT
  reduce(D)(grS | noDanglingRefs?(grS)) HAS_TYPE (noDanglingRefs?)

reduce_arm: JUDGEMENT
  reduce(D)(grS | noDanglingRefs?(grS) AND allReleaseMarked?(grS`redex) AND armc?(grS`context))
    HAS_TYPE { rS | allReleaseMarked?(rS`redex) AND armc?(rS`context) }
```

The definition of the *matching* relation for the bisimulation is as follows, where `translate_refs` is the operation that translates all the references in its second argument with respect to the translation function given as first argument.

```
stack_matches(translate)(S, S1, X): boolean =
  S`length = S1`length AND
    FORALL (i: (X)):
      (i < S`length AND
      translate_refs(translate)(S`seq(S`length - i - 1)) =
        S1`seq(S1`length - i - 1))

store_matches(translate)(rS, dom1, (store1: Store(dom1))): boolean =
  FORALL (r: (rS`domain)):
    rS`count(r) > 0 IMPLIES
      dom1(translate(r)) AND store1(translate(r))`length = rS`store(r)`length AND
        FORALL (j: below(rS`store(r)`length)):
          translate_refs(translate)(rS`store(r)`seq(j)) = store1(translate(r))`seq(j)

state_matches(eS, rS)(translate): boolean =
      eS`error = rS`error
    AND translate_refs(translate)(unmark(rS`redex)) = eS`redex
    AND unmark(rS`context) = eS`context
    AND stack_matches(translate)(rS`stack, eS`stack,
          union(cvars(rS`redex), bumpn(cvars(rS`context), popDepth(rS`redex))))
    AND store_matches(translate)(rS, eS`domain, eS`store)

state_matches?(eS, rS): boolean =
  EXISTS (translate): state_matches(eS, rS)(translate)
```

Finally we can prove Theorems 3 and 4 as follows.

```
bisimulation_lemma: THEOREM
  noDanglingRefs?(grS) AND allReleaseMarked?(grS`redex) AND armc?(grS`context) AND
      state_matches?(to_topstate(gS), to_topstate(grS)) IMPLIES
    state_matches?(reduce(D)(gS), rreduce_n(D)(top_releases_ct(grS`redex) + 1, grS))

bisimulation_theorem: THEOREM
  noDanglingRefs?(grS) AND allReleaseMarked?(grS`redex) AND armc?(grS`context) AND
      state_matches?(gS, grS) IMPLIES
    EXISTS (n: posnat): state_matches?(reduce(D)(gS), rreduce_n(D)(n, grS))
```

For *IL*, the datatypes are as follows:

```
iexpr: DATATYPE
BEGIN
  ivar(vindex: nat, vmarked: bool): ivar?
  iconstant(ival: int): iconstant?
  inil: inil?
  icall(ifun: nat, iargs: list[(ivar?)]): icall?
  iupdate(itarget, ilhs, irhs: (ivar?)): iupdate?
  ilookup(iaval, ipos: (ivar?)): ilookup?
  inewint(size: nat): inewint?
  inewref(size: nat): inewref?
END iexpr

istat: DATATYPE
BEGIN
  iassign(avar: (ivar?), aexpr: iexpr): iassign?
  idecl(dtype: nat, dstat: istat): idecl?
  iif(icond: (ivar?), iftrue, iffalse: istat): iif?
  iskip: iskip?
  iblock(block1, block2: istat): iblock?
  irelease(rvar: (ivar?)): irelease?
END istat
```

As previously, we use a different function for each possible reduction; for instance, here is one example reduction (`endi` is a function which indexes finite sequences from the end, so that they can be used as stacks). It corresponds to the beginning of a block, i.e., the 5th case in Figure 12.

```
get_frame_body(iS | iS`callstack`length > 0): [stack_frame(iS`defs), istat] =
  (endi(iS`callstack, 0), iS`defs`seq(endi(iS`callstack, 0)`ffct)`body)

pushReduce(iS | iS`callstack`length > 0 AND pushRedex?(iS)): istate =
  LET (lf, fbody) = get_frame_body(iS) IN
  iS WITH [`stack := add(vundef, iS`stack),
           `callstack`seq(iS`callstack`length - 1) :=
             lf WITH [
               `fdepth := lf`fdepth + 1,
               `fpc := next_pc(fbody, lf`fpc)
             ]
          ]
```

Finally, to define the bisimulation relation for Theorem 5, the following declarations are used. The definitions `store_matches` and `count_matches` check that the store and the count, respectively, are the same between the two reductions, while `redex_matches` checks that the redex and the stack are matching between the two versions.

```
store_matches(dom, (str1: [(dom) -> finseq[(value?)]]), (str2: [(dom) -> finseq[ivalue]])): bool =
  FORALL (i: (dom)):
    str1(i)`length = str2(i)`length AND
    FORALL (j: below(str1(i)`length)):
      str2(i)`seq(j) = val_to_ival(str1(i)`seq(j))

count_matches(dom, (cnt1, cnt2: [(dom) -> int])): bool =
  FORALL (i: (dom)): cnt1(i) = cnt2(i)

redex_matches(D, (trS | defs_well_typed(D, trS`def_types)),
  (iS | iS`defs = translate_definitions(D, trS`def_types) AND iS`callstack`length > 0)): bool =
  LET fv = endi(iS`callstack, 0) IN
  LET (A, K, tv) = fct_reconstruct_state(D`seq(fv`ffct)`body, iS`stack,
                     D`seq(fv`ffct)`arity, fv`fdepth, fv`fpc) IN
  LET K1 = extract_call_context(D, trS`def_types, iS) IN
  LET tv1 = extract_call_tv(D, trS`def_types, iS) IN
  LET tv2 = compose_translate_vars(tv, tv1, popDepth(K) + popDepth(A), fv`fdepth) IN
  fill(compose(K1, K), A) = fill(trS`state`context, trS`state`redex) AND
  (FORALL (i: below(popDepth(trS`state`redex) + popDepth(trS`state`context))):
    endi(iS`stack, tv2(i)) = val_to_ival(endi(trS`state`stack, i))) AND
  fct_is_result_defined(D`seq(fv`ffct)`body, iS`stack, D`seq(fv`ffct)`arity, fv`fdepth, fv`fpc)

state_matches(D, (trS | defs_well_typed(D, trS`def_types)), iS): bool =
  iS`defs = translate_definitions(D, trS`def_types) AND
  iS`domain = trS`state`domain AND
  store_matches(trS`state`domain, trS`state`store, iS`store) AND
  count_matches(trS`state`domain, trS`state`count, iS`count) AND
  IF iS`callstack`length = 0 THEN
    trS`state`context = hole AND
      value?(trS`state`redex) AND
      iS`stack`seq(0) = val_to_ival(trS`state`redex)
  ELSE
    redex_matches(D, trS, iS)
  ENDIF AND
  trS`state`error = iS`error
```

Here, `val_to_ival` converts between the two different types of values and `translate_definitions` translates all definitions using the algorithm in Figure 13. Since we use de Bruijn indices, it is not the variables names but their indices which matter, and these are different in the original and the translated version, so the task of the `tv`, `tv1` and `tv2` variables is to make this translation, while `extract_call_tv` extracts this translation for all the functions in the call stack except the current one, and `compose_translate_vars` reconstructs the global translation function. Finally, `fct_translate_state` and `extract_call_context` respectively extract the redex and context from the current function and the context from the rest of the call stack, while `fct_is_result_defined` checks that if we are after the place where the result variable should have been defined, then it is not bound to `undef` (which would break the correctness of redex extraction).

We can then prove the result below.

```
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))
```

All of the code is available at `https://github.com/SRI-CSL/PVSCodegen`.

6 Conclusions

The PVS2C code generator [2] translates an applicative fragment of PVS into C code. This work proves the correctness of an idealized version of this code generator, translating a functional language that is a subset of the intermediate language of PVS2C code generator to an imperative language with high-level operations for reference counting.

Future work could include verifying the translation of that imperative language to a subset of C, the translation of a fragment of PVS to the intermediate language, and supporting first-class functions and closures. Once the translation to C is verified as well, certified C compilers such as CompCert [4] can be used to achieve fully certified code generation.

References

[1] M. Felleisen. On the expressive power of programming languages. In *European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 35–75. Springer-Verlag, 1990.

[2] G. Férey and N. Shankar. Code generation using a formal model of reference counting. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 150–165, Cham, 2016. Springer International Publishing.

[3] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 502–514. ACM, 1993.

[4] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.