The Correctness of a Code Generator for a Functional Language

Nathanaël Courant[1], Antoine Séré[2], and Natarajan Shankar[3]

[1] Inria Paris and Université Paris Diderot, Paris, France
[2] École Polytechnique, Palaiseau, France
[3] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

Abstract. Code generation is gaining popularity as a technique to bridge the gap between high-level models and executable code. We describe the theory underlying the PVS2C code generator that translates functional programs written using the PVS specification language to standalone, efficiently executable C code. We outline a correctness argument for the code generator. The techniques used are quite generic and can be applied to transform programs written in functional languages into imperative code. We use a formal model of reference counting to capture memory management and safe destructive updates for a simple first-order functional language with arrays. We exhibit a bisimulation between the functional execution and the imperative execution. This bisimulation shows that the generated imperative program returns the same result as the functional program.

1 Introduction

Functional languages offer a convenient and expressive notation for defining programs in a form that is referentially transparent and amenable to mathematical proof. One way of implementing a functional language on a machine is to transform a given program into a corresponding program in an imperative programming language. There are two key challenges in defining such a transformation. One, the evaluation of expressions in a functional language is pure, so that updating an array creates a fresh copy of the array being updated. Replacing such an update with a destructive, in-place update is not always sound. Two, allocated structures like arrays have to be garbage collected when they are no longer referenced in the evaluation.

We are interested in proving the correctness of a transformation from a functional program to a self-contained imperative program that executes efficiently and performs its own memory management. To this end, we use a transformation that employs reference counting for managing memory as well as for identifying opportunities for safe destructive updates during execution. The transformation is enabled by a light static analysis on the input functional program. This analysis helps release references as soon as possible in order to maximize the opportunities for destructive updates. We present a proof method for demonstrating the correctness of the transformation and a formalization of the correctness of a

transformation from a small functional language to a C-like imperative language. The transformation from functional to self-contained imperative code forms the core of the PVS2C code generator [10,23]. Such transformations and the intermediate languages used in them are of foundational interest and practical utility for the generation of efficient code from executable fragments of specification and modelling languages.

Since code generators are becoming increasingly popular, it is important to ensure that they can be backed with simple, easily verifiable correctness proofs. The correctness of the transformation from a functional to an imperative language is carried out in multiple steps. The source language *FL* for our code generator is an idealized first-order functional language where programs are written in A-normal form [11]. This language is lightly typed and can serve as an intermediate language for multiple source languages. The operational semantics is presented in terms of reductions within an evaluation context [9]. This semantics is pure: each array update allocates a new array and copies the contents of the old array and performs the update on the copy.

We next define the operational semantics of an annotated variant *RL* of *FL* that exploits reference counting and destructive updates. We exhibit a bisimilarity between the *FL* and *RL* operational semantics so that these two forms of evaluation always yield the same value, when either evaluation terminates. The correspondence between *FL* and *RL* has been already been verified in PVS.

Next, we define a translation from annotated *FL* to an imperative language *KL*. The latter language is inspired by the operational semantics given by Appel and Blazy [1]. The language *KL* uses explicit assignments and the operational semantics employs continuations so that there is a significant semantic gap between *RL* executions and their *KL* counterparts. Even so, we exhibit a bisimulation between the operational semantics for the reference counting execution of *RL* and that of the imperative language *KL*.[4]

We give a brief overview of languages, the code generator, and the correctness arguments. Consider the *FL* program *swap* which swaps two elements of an array.

$$\begin{aligned}\texttt{swap}(u,i,j) = \ & \texttt{let}\ a = u[i] \\ & \texttt{in let}\ b = u[j] \\ & \quad \texttt{in let}\ u' = u[i \mapsto b] \\ & \quad\quad \texttt{in}\ u'[j \mapsto a]\end{aligned}$$

The body of the definition is in A-normal form [11]. The array access and update operations are applied to arguments that are variables. Our A-normal form is unflattened so that the expressions e_1 and e_2 in a let-expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ are both recursively in A-normal form.

[4] We had initially used a different semantics for the imperative language based on call stacks and program counters that is closer to the machine execution, but this led to a fairly cumbersome definition of the bisimulation. We found the mechanization (`https://github.com/SRI-CSL/PVSCodegen`) of the correspondence quite challenging. The correspondence given here between *RL* and *KL* executions has not yet been formalized using a proof assistant, but we expect it to be a significantly easier exercise.

Given a body of definitions Δ of the above form, we would like to evaluate an expression given in A-normal form. The evaluation of an expression e can be carried out with respect to a stack S which binds variables to values (integers or references) and a store which maps references to arrays. For example if the expression e is $\texttt{let}\ x = y[i \mapsto k]\ \texttt{in}\ x[i]$, where the subexpression $y[i \mapsto k]$ denotes the result of updating the array y at i with k. Let us assume that we are evaluating e with a stack S of the form $\langle y \mapsto r_0, i \mapsto 2, k \mapsto 5\rangle$ and a store $\mathcal{M}$ given by the map $\{r_0 \mapsto \mathbf{A}\}$, where $\mathbf{A}$ is the array $\langle 1, 2, 3\rangle$. The expression e can be viewed as an evaluation context $\texttt{let}\ x = \Box\ \texttt{in}\ x[i]$ with a single hole $\Box$, and a redex $y[i \mapsto k]$ filling the hole. The operational semantics is given as a set of rewrite rules on the triple $(d, S, \mathcal{M})$ for redex d, stack S, and store $\mathcal{M}$. In this case, the reduction yields a reference r_1 and a new store $\mathcal{M}'$ that extends $\mathcal{M}$ with the map $\{r_1 \mapsto \langle 1, 2, 5\rangle\}$. The new state now has the expression $\texttt{let}\ x = r_1\ \texttt{in}\ x[i]$, which contains an explicit reference, namely, the reference resulting from the prior reduction. This expression is a redex by itself. Reducing this redex with stack S and store $\mathcal{M}'$ yields the expression $\texttt{pop}(x[i])$, the new stack $\langle x \mapsto r_1, y \mapsto r_0, i \mapsto 2, k \mapsto 5\rangle$, and the same store $\mathcal{M}$. The operation `pop` is a book-keeping operation used to pop the stack at the end of the evaluation of the body of the let-expression. The subexpression $x[i]$ is a redex and reduces to 5, and finally the redex $\texttt{pop}(5)$ is reduced by popping the binding for x off the stack to yield the value 5 and the resulting stack S and store $\mathcal{M}'$.

The reference counting semantics maintains a count for each reference in the domain of the store $\mathcal{M}$. The *RL* expression being evaluated is annotated so that the last lexical occurrence of a variable along any evaluation path is marked (i.e., underlined). For example, the expression e above would be annotated as $\texttt{let}\ x = \underline{y}[i \mapsto \underline{k}]\ \texttt{in}\ \underline{x}[\underline{i}]$. If we evaluate this expression with a stack S as above, and a store $\mathcal{M}$ of the form $\{r_0 \mapsto [1, 2, 3]\}$, and a reference count C of the form $\{r_0 \mapsto 1\}$. With this, the redex $\underline{y}[i \mapsto \underline{k}]$ is reduced to r_0, the new stack S' is of the form $\langle y \mapsto \texttt{nil}, i \mapsto 2, k \mapsto 5\rangle$, the new store $\mathcal{M}'$ is just $\{r_0 \mapsto \langle 1, 2, 5\rangle\}$, and the new reference count C' is $\{r_0 \mapsto 1\}$. In other words, we can perform the update in place since the variable y is marked and its reference count is 1, indicating that there are no further uses of the reference r_0 in the evaluation nor for the stack binding of the variable y in the stack. If the variable y is unmarked or the corresponding reference r_0 has a count greater than 1, then we need to create a (shallow) copy of the array before updating it. The next step of evaluation binds x to r_0 in the stack and continues executing as above. When the redex $\underline{x}[\underline{i}]$ is evaluated, the reference count for r_0 becomes 0, and the array is freed. Here, the array contains integers, but if the array being freed contains references, these references need to have their reference counts decremented. The bisimulation between the pure evaluation and the reference counting destructive evaluation shows that the reference count is tracked accurately. It also shows that there is a map from the references in the destructive evaluation to those in the pure evaluation such that the corresponding expressions, stacks, and stores match.

The next step in our proof transforms the annotated language *RL* to an imperative language *KL*. The imperative language, by design, looks quite similar

to the functional language but employs assignments. The translation of the expression $\texttt{let}\ x = \underline{y}[i \mapsto \underline{k}]\ \texttt{in}\ \underline{x}[\underline{i}]$ into *KL* is done in the context of a result variable `return`. In *IL*, we get a program $\{\texttt{int}\ x; x := \underline{y}[i \mapsto \underline{k}]; \texttt{return} := \underline{x}[\underline{i}]\}$.

Like *RL*, the execution of the imperative language *KL* tracks reference counts and uses the marking to release references. A reference is released by decrementing its reference count and freeing memory when the reference count drops to 0, but only after recursively releasing the references in the contents of the array. In addition to the stack $\mathcal{S}$, the store $\mathcal{M}$, and the reference count $\mathcal{C}$, the operational semantics for *KL* maintains a continuation $\mathcal{K}$ that is just a program representing the rest of the computation. Let the initial continuation $\mathcal{K}_0$ be $\{\texttt{int}\ x; x := \underline{y}[i \mapsto \underline{k}]; \texttt{return} := \underline{x}[\underline{i}]\}$, the initial stack $\mathcal{S}_0$ be $\langle \texttt{return} \mapsto \texttt{undef}, i \mapsto 2, k \mapsto 5, y \mapsto r \rangle$, the initial store $\mathcal{M}_0$ be $\{r_0 \mapsto \langle 1, 2, 3 \rangle\}$, and the initial reference count table $\mathcal{C}_0$ be $\{r_0 \mapsto 1\}$. The declaration $\texttt{int}\ x$ is evaluated by extending the stack with the binding $x \mapsto \texttt{undef}$ so that $\mathcal{S}_1$ is $\langle x \mapsto \texttt{undef}, \texttt{return} \mapsto \texttt{undef}, i \mapsto 2, k \mapsto 5, y \mapsto r_0 \rangle$, while appending a `pop` instruction to the right of the continuation corresponding to the binding for x to yield $\mathcal{K}_1 = x := \underline{y}[i \mapsto \underline{k}]; \texttt{return} := \underline{x}[\underline{i}]; \texttt{pop}$. The store and reference count tables are left unchanged in $\mathcal{M}_1$ and $\mathcal{C}_1$. The assignment $x := \underline{y}[i \mapsto \underline{k}]$ is executed by evaluating the right-hand side by updating the array bound to r_0 in the store, binding x in the stack to the reference r_0, and releasing the binding to y. Now, $\mathcal{K}_2$ is $\texttt{return} := \underline{x}[\underline{i}]; \texttt{pop}$, $\mathcal{S}_2$ is $\langle x \mapsto r_0, \texttt{return} \mapsto \texttt{undef}, i \mapsto 2, k \mapsto 5, y \mapsto \texttt{nil} \rangle$, $\mathcal{M}_1$ is $\{r_0 \mapsto \langle 1, 2, 5 \rangle\}$, and $\mathcal{C}_1$ is $\{r_0 \mapsto 1\}$. The assignment to `return` is then executed and the remaining `pop` instructions are executed to yield the final state where $\mathcal{K}_4$ is empty, the $\mathcal{S}_4$ is set to $\langle \texttt{return} \mapsto 5, i \mapsto 2, k \mapsto 5, y \mapsto \texttt{nil} \rangle$, and $\mathcal{M}_4$ and $\mathcal{C}_4$ are both empty. The bisimulation between the evaluation of an *RL* expression and the execution of its translation in *KL* is quite challenging since there are subtle semantic differences between the executions of these two languages.

The goal of our proof exercise is to construct a simple and elegant formalization of the correctness of the correspondence between the evaluations of the source and target of a code generator that is amenable to easy mechanical verification. We have defined intermediate representations that simplify the proofs while retaining the flexibility to support multiple source languages and target multiple imperative languages. The proofs have been designed so that the language can be extended with new features with minimal impact on the invariants and bisimulations. The formally defined code generator presented here is an idealization of a practical code generator from a functional language to C. This code generator produces readable, self-contained C code with a modest overhead for reference counting.

Related Work. Reference counting was introduced by Collins [5] in 1960. It was shown to fail in the presence of cyclic structures by McBeth [18] in 1963. Reference cycles cannot appear in the execution of PVS or in any of the languages *FL*, *RL*, or *KL* so that our use of reference counting is sound. There are several proofs of the correctness of reference counting implementations and reference counting garbage collectors [8,19]. Hudak [13] presents an abstract interpretation in terms reference counts as a way of optimizing program execution.

Several papers present static analyses for safe destructive updates in functional languages [14,7,12,26,22,2]. Chirimar, Gunter, and Riecke [4] define a reference counting abstraction machine for a computational interpretation of linear logic. The work that is closest to our own is Schulte's code generator [21] for the specification language Opal [6] that translates a first-order functional fragment of the language into a reference counted implementation in C in which execution interacts with the garbage collector to reuse storage (see also de Moura and Ullrich [25]). The analysis and transformations used here are similar though the intermediate languages and proof techniques are quite different. We are using a formal model of reference counting to dynamically manage memory for a source language with object updates that can be executed destructively when safe. The presentation here is the basis for a practical implementation of a code generator that covers a full functional language with arrays, records, tuples, algebraic datatypes, and closures, as well as the outline for a machine-verified proof for its correctness.

Formal verification of compilers is a well-studied topic [20,3,24,15,17,16]. This paper presents the theory underlying a simple code generator. Based on our prior experience, we estimate that the mechanization of the proofs here would require fewer than ten person-weeks, whereas the correctness of the full PVS2C code generator would involve a substantial months-long effort. In contrast, proving the correctness of a compiler is a much larger undertaking.

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 simplicity; however, in the paper we will use identifiers when giving examples to make the example more readable. The syntax of *FL* is defined in Figure 1a. The functions f can be primitive functions like $+$, $*$, and $-$, or defined functions. The sequence of variables $x_1, \ldots, x_n$ in a function application $f(x_1, \ldots, x_n)$ may contain duplicates.

The variables and operations have types associated with them in order to simplify the definition of various memory management operations. The expression $vars(e)$ is defined as the set of free variables in e. A type t is either the integer type `int` or is an n-element array $t[n]$ with element type t. The constant `nil` is seen as a valid constant for any array type of the form $t[n]$.

$$t ::= \texttt{int} \mid t[n]$$

Each function symbol has a type $t_1 \times \ldots \times t_n \to t$, where the i'th argument has type t_i, and the range type is t. Given the types of the variables and function symbols, the type of a well-formed expression e can be computed as $\tau(e)$.

The type system does not rule out errors due to null dereferencing and out-of-bounds array access. Given the types for variables and operations, each well-typed expression has a unique type. The type rules are straightforward and are left as an exercise for the reader. For the sake of simplicity, we will often

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

(a) Syntax of *FL*

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

(b) Definition of a redex

Fig. 1: Expression and Redex Syntax of *FL*

omit the type annotations on the `let` constructors. Note that the `pop` and `ref` constructors are not allowed in programs and are used only during reduction. We also restrict the primitive functions to operate solely over the integers.

We also define an evaluation context with a hole $\Box$ marking the location where evaluation occurs, as below.

$$K ::= \Box \mid \texttt{let}\ (x : t) = K_1\ \texttt{in}\ e_1 \mid \texttt{pop}(K_1)$$

The composition $K[e]$ of a context K and an expression e consists in replacing the hole $\Box$ in K by the expression e. A *value* is a reference, a constant or `nil`. A *redex* is defined in Figure 1b.

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

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. The empty stack is written as *empty*. Entries A are pushed on to the stack by the operation $\mathbf{push}(A, S)$. When S is a stack of variable bindings, a binding $x \mapsto v$ is pushed on to the stack by the operation $\mathbf{push}(x \mapsto v, S)$. For a sequence of variables $x_1, \ldots, x_n$ (abbreviated as $\overline{x}$) and values $v_1, \ldots, v_n$ (abbreviated as $\overline{v}$), we abbreviate $\mathbf{push}(x_n \mapsto v_n, \ldots, \mathbf{push}(x_1 \mapsto v_1, S)\ldots)$ as $\mathbf{push}(\overline{x} \mapsto \overline{v}, S)$. The lookup operation $S(x)$ retrieves the topmost binding for x. This operation is only invoked when there is a binding for x in S. We also use the operation $S[x \mapsto v]$ to update the topmost binding for x in the stack. This operation is also applied only when there is a binding for x in S. We denote by $\mathbf{new}(\mathcal{M})$, a reference that is not yet defined in $\mathcal{M}$. We use $f(v_1, \ldots, v_n) \xrightarrow{\delta} v$ for the reduction relation

$$
\begin{aligned}
(x, S, \mathcal{M}) &\to (S(x), S, \mathcal{M}) \\
(x[y], S, \mathcal{M}) &\to (\mathcal{M}(S(x))(S(y)), S, \mathcal{M}) \\
(x[y \mapsto z], S, \mathcal{M}) &\to (r, S, \mathcal{M}[r \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]]), \\
&\qquad \text{where } r = \mathbf{new}(\mathcal{M}) \\
(\texttt{newint}(n), S, \mathcal{M}) &\to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \langle 0, \ldots, 0\rangle]) \\
(\texttt{newref}(n), S, \mathcal{M}) &\to (\mathbf{new}(\mathcal{M}), S, \mathcal{M}[\mathbf{new}(\mathcal{M}) \mapsto \langle \texttt{nil}, \ldots, \texttt{nil}\rangle]) \\
(\texttt{let } x = v \texttt{ in } e, S, \mathcal{M}) &\to (\texttt{pop}(e), \mathbf{push}(x \mapsto v, S), \mathcal{M}) \\
(\texttt{pop}(v), S, \mathcal{M}) &\to (v, \mathbf{pop}(S), \mathcal{M}) \\
(\texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2, S, \mathcal{M}) &\to \begin{cases} (e_1, S, \mathcal{M}), & \text{if } S(x) \neq 0 \\ (e_2, S, \mathcal{M}), & \text{otherwise} \end{cases} \\
(f(x_1, \ldots, x_n), S, \mathcal{M}) &\to (v, S, \mathcal{M}) \text{ for primitive } f, \\
&\qquad \text{where } f(S(x_1), \ldots, S(x_n)) \xrightarrow{\delta} v \\
(f(x_1, \ldots, x_n), S, \mathcal{M}) &\to (\texttt{pop}^n(e), \mathbf{push}(\overline{y} \mapsto S(\overline{x}), S), \mathcal{M}) \\
&\qquad \text{where } f(y_1, \ldots, y_n) = e \text{ in } \Delta.
\end{aligned}
$$

Fig. 2: Operational Semantics of *FL*

capturing the evaluation of primitive functions such as $+$. The non-primitive functions are defined in the program Δ which maps the function symbol f to its definition of the form $f(y_1, \ldots, y_n) = e$ where $vars(e) = \{y_1, \ldots, y_n\}$, and $y_i \neq y_j$ for $1 \leq i < j \leq n$.

The small-step semantics are defined as the unique context-preserving relation $\to$ that is defined on redexes as in Figure 2. It is easy to see that the reductions are deterministic. An evaluation step has the form $(E[e], S, \mathcal{M}) \longrightarrow (E[e'], S', \mathcal{M}')$ iff $(e, S, \mathcal{M}) \to (e', S', \mathcal{M}')$.

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

Let $\texttt{swap}(u, i, j)$ be defined as

$$\texttt{let } a = u[i] \texttt{ in let } b = u[j] \texttt{ in let } u' = u[i \mapsto b] \texttt{ in } u'[j \mapsto a].$$

Given $e = \texttt{let } z = +(y, 1) \texttt{ in swap}(x, y, z)$ with $S = (y \mapsto 0, x \mapsto r)$ and $\mathcal{M} = (r \mapsto \langle 0, 1\rangle)$, we show the steps in the reduction of $(e, S, \mathcal{M})$ in Figure 3.

3 Evaluation with Reference Counting

The reference-counting language *RL* extends *FL* with an additional constructor, $\texttt{release}(x, e)$, which is also a redex. In addition, a variable occurrence in *RL*

$$\begin{pmatrix}\texttt{let } z = +(y,1) \texttt{ in } \texttt{swap}(x,y,z),\\ (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)\end{pmatrix} \tag{1}$$

$$\longrightarrow (\texttt{let } z = 1 \texttt{ in } \texttt{swap}(x,y,z), (y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)) \tag{2}$$

$$\longrightarrow (\texttt{pop}(\texttt{swap}(x,y,z)), (z \mapsto 1, y \mapsto 0, x \mapsto r), (r \mapsto \langle 0,1\rangle)) \tag{3}$$

$$\longrightarrow \begin{pmatrix}\ldots \texttt{let } a = u[i] \texttt{ in } \ldots,\\ (j \mapsto 1, i \mapsto 0, u \mapsto r, \ldots),\\ (r \mapsto \langle 0,1\rangle)\end{pmatrix} \tag{4}$$

$$\longrightarrow \begin{pmatrix}\ldots \texttt{let } a = 0 \texttt{ in } \ldots,\\ (j \mapsto 1, i \mapsto 0, t \mapsto r, \ldots), (r \mapsto \langle 0,1\rangle)\end{pmatrix} \tag{5}$$

$$\longrightarrow \ldots \texttt{let } b = u[j] \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots), (r \mapsto \langle 0,1\rangle)) \tag{6}$$

$$\longrightarrow (\ldots \texttt{let } b = 1 \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots), (r \mapsto \langle 0,1\rangle)) \tag{7}$$

$$\longrightarrow \begin{pmatrix}\ldots \texttt{let } u' = u[i \mapsto b] \texttt{ in } \ldots,\\ (b \mapsto 1, a \mapsto 0, \ldots),\\ (r \mapsto \langle 0,1\rangle)\end{pmatrix} \tag{8}$$

$$\longrightarrow \begin{pmatrix}\ldots \texttt{let } u' = r' \texttt{ in } \ldots,\\ (b \mapsto 1, a \mapsto 0, \ldots),\\ (r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle)\end{pmatrix} \tag{9}$$

$$\longrightarrow (\ldots u'[j \mapsto a] \ldots, (u' \mapsto r', b \mapsto 1, \ldots), (r' \mapsto \langle 1,1\rangle, \ldots)) \tag{10}$$

$$\longrightarrow (\texttt{pop}^7(r''), (u' \mapsto r', b \mapsto 1, \ldots), (r'' \mapsto \langle 1,0\rangle, \ldots)) \tag{11}$$

$$\xrightarrow{+} (r'', (y \mapsto 0, x \mapsto r), (r'' \mapsto \langle 1,0\rangle, r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle)) \tag{12}$$

Fig. 3: An example *FL* reduction

can be *marked* to indicate that this is the last occurence of the variable along an evaluation path. The evaluation state is extended with $\mathcal{C}$ which maps each reference to its reference count.

Define $\#(S,r)$ as the number of times the reference r occurs as a binding in stack S, $\#(a,r)$ as the number of occurrences of r as an element of the array a, and $\mathbb{1}_{r\in e}$ as 1 if r occurs as a value in e, and 0, otherwise. The key invariant that is maintained is that the value of $\mathcal{C}(\texttt{ref}(k))$ is exactly the reference count of $\texttt{ref}(k)$ in e, S, and $\mathcal{M}$.

$$\mathcal{C}(\texttt{ref}(k)) = \mathbb{1}_{\texttt{ref}(k)\in e} + \#(S, \texttt{ref}(k)) + \sum_{\texttt{ref}(j)\in\mathcal{M}} \#(\mathcal{M}(\texttt{ref}(j)), \texttt{ref}(k)) \tag{13}$$

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

The evaluation also preserves three key 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 re-marking the result (using the **mark** algorithm shown below) returns an expression identical to e).

release-marked. All subterms of e of the form $\texttt{release}(x, e')$ have the occurrence of x in the first argument marked.

We use the following helper functions:

$$\begin{aligned}
\mathbf{incr}(\texttt{ref}(k), \mathcal{C}) &= \mathcal{C}[\texttt{ref}(k) \mapsto \mathcal{C}(v) + 1] \\
\mathbf{incr}(v, \mathcal{C}) &= \mathcal{C} && \text{otherwise} \\
\mathbf{decr}(\texttt{ref}(k), \mathcal{C}) &= \mathcal{C}[\texttt{ref}(k) \mapsto \mathcal{C}(v) - 1] \\
\mathbf{decr}(v, \mathcal{C}) &= \mathcal{C} && \text{otherwise}
\end{aligned}$$

The function **decref** 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 1, it recursively (using $\mathbf{decref}\star$) decreases the count of all the non-`nil` references pointed by that one and replaces them by `nil` before freeing the memory allocated to a reference r by setting $\mathcal{M}(r)$ to $\bot$. The termination of the mutually recursive definitions of **decref** and $\mathbf{decref}\star$ is given by a lexicographic measure on the size of the type t and the array index m.[5]

$$\begin{aligned}
\mathbf{decref}(t[n])(\texttt{ref}(k), (\mathcal{M}, \mathcal{C})) &= (\mathcal{M}, \mathbf{decr}(\texttt{ref}(k), \mathcal{C})), \text{ if } C(\texttt{ref}(k)) > 1 \\
\mathbf{decref}(t[n])(\texttt{ref}(k), (\mathcal{M}, \mathcal{C})) &= (\mathcal{M}'[\texttt{ref}(k) \mapsto \bot], \mathbf{decr}(\texttt{ref}(k), \mathcal{C}')), \\
&\quad \text{if } C(\texttt{ref}(k)) = 1 \\
&\quad \text{where } (\mathcal{M}', \mathcal{C}') = \mathbf{decref}\star(t)(v, (\mathcal{M}, \mathcal{C}), n) \\
\mathbf{decref}(t)(v, (\mathcal{M}, \mathcal{C})) &= (\mathcal{M}, \mathcal{C}), \text{ otherwise} \\
\mathbf{decref}\star(t[n])(\texttt{ref}(k), (\mathcal{M}, \mathcal{C}), m+1) &= \mathbf{decref}(t)(\mathcal{M}(\texttt{ref}(k))[m], (\mathcal{M}'', \mathcal{C}')), \\
&\quad \text{where } (\mathcal{M}', \mathcal{C}') = \mathbf{decref}\star(t[n])(r, (\mathcal{M}, \mathcal{C}), m) \\
&\quad\ \ \mathcal{M}'' = \mathcal{M}'[r \mapsto \mathcal{M}'(r)[m \mapsto \texttt{nil}]] \\
&\quad\ \ r = \texttt{ref}(k) \\
\mathbf{decref}\star(t[n])(\texttt{ref}(k), (\mathcal{M}, \mathcal{C}), 0) &= (\mathcal{M}, \mathcal{C}) \\
\mathbf{decref}\star(t)(v, (\mathcal{M}, \mathcal{C}), m) &= (\mathcal{M}, \mathcal{C}), \text{ otherwise}
\end{aligned}$$

Expressions being evaluated are analyzed in order to mark the last occurrence of a variable along any evaluation path. This marking helps to identify the lifetime of the variable by indicating the point at which the variable is no longer

[5] Note that due to the recursion on type structure, the termination proofs do not need to assume that the store is non-cyclic. In our mechanization, we use a slightly different definition and exploit Invariant 13 and the invariant (also implicit in **decref**) that $\mathcal{M}$ contains no (dangling) references that are not in the domain of $\mathcal{M}$ so that the total reference count in $\mathcal{M}$ decreases with each call to **decref**.

used in a computation. The operation $\mathbf{mark}(X, e)$, a few cases of which are defined below, marks each variable in e that is not in X. In the remaining cases, $\mathbf{mark}(X, e)$ marks the last (non-binding) occurrence in e of any variable in $vars(e) - X$. We overload `release` so that $\texttt{release}(\{x_1, \ldots, x_n\}, e)$ is shorthand for $\texttt{release}(x_n, \ldots \texttt{release}(x_1, e) \ldots)$.

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

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

$$\mathbf{mark}(X, \texttt{ifnz } x \texttt{ then } e_1 \texttt{ else } e_2) = \begin{cases} \texttt{ifnz } \mathbf{mark}(\mathbf{vars}(e_1) \cup \mathbf{vars}(e_2) \cup X, x) \\ \quad \texttt{then release}(\mathbf{vars}(e_2) - (X \cup \mathbf{vars}(e_1)), \mathbf{mark}(X, e_1)) \\ \quad \texttt{else release}(\mathbf{vars}(e_1) - (X \cup \mathbf{vars}(e_2)), \mathbf{mark}(X, e_2)) \end{cases}$$

For example, $\mathbf{mark}(\emptyset, \texttt{let } x = f(y) \texttt{ in ifnz } z \texttt{ then } g(x, y) \texttt{ else } f(x))$ is

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

We translate *FL* programs into marked *RL* programs by replacing each definition of the form $f(\overline{y}) = e$ in Δ by $f(\overline{y}) = \mathbf{mark}(\emptyset, e)$ in the *RL* program $\Delta^{\#}$.

Given a sequence $x_1, \ldots, x_n$ (abbreviated as $\overline{x}$) of (possibly marked) variables. Let $\mathbf{incvars}(\overline{x}, S, \mathcal{C})$ represent the result of incrementing the count of $S(y)$ by one for each unmarked variable y in $\overline{x}$.

$$\begin{aligned} \mathbf{incvars}(\langle\rangle, S, \mathcal{C}) &= \mathcal{C} \\ \mathbf{incvars}((x_1, x_2, \ldots, x_n), S, \mathcal{C}) &= \mathbf{incvars}((x_2, \ldots, x_n), S, \mathcal{C}'), \text{ where} \\ \mathcal{C}' &= \begin{cases} \mathbf{incr}(S(x), \mathcal{C}), & \text{if } x \text{ is unmarked} \\ \mathcal{C}, & \text{otherwise .} \end{cases} \end{aligned}$$

Figure 4 shows a few cases of the definition of reduction for *RL* redexes.

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, let $\texttt{swap}(u, i, j)$ be defined as

$$\texttt{let } a = u[i] \texttt{ in let } b = u[j] \texttt{ in let } u' = \underline{u}[\underline{i} \mapsto \underline{b}] \texttt{ in } \underline{u'}[\underline{j} \mapsto \underline{a}].$$

$$
\begin{aligned}
(x, S, (\mathcal{M}, \mathcal{C})) &\to^{\#} (S(x), S[x \mapsto \texttt{nil}], (\mathcal{M}, \mathcal{C})) \\
&\quad \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}, \mathbf{incvars}(x, \mathcal{M}, \mathcal{C}))) \text{ otherwise} \\
(f(\overline{x}), S, (\mathcal{M}, \mathcal{C})) &\to^{\#} (e, S_n, (\mathcal{M}, \mathcal{C}')), \text{ where} \\
&\quad \mathcal{C}' = \mathbf{incvars}(\overline{x}, S, \mathcal{C}) \\
&\quad S_0 = \mathbf{push}(\overline{y} \mapsto S(\overline{x}), S) \\
&\quad S_{i+1} = \begin{cases} S_i[x_i \mapsto \texttt{nil}], \\ \text{ if } x_i \text{ is a marked and } S(x) \text{ is a reference} \\ S_i, \text{ otherwise} \end{cases} \\
(x[y \mapsto z], S, (\mathcal{M}, \mathcal{C})) &\to^{\#} (S(x), S'[x \mapsto \texttt{nil}], (\mathcal{M}'', \mathcal{C}'')), \\
&\quad \text{if } \mathcal{C}(S(x)) = 1 \text{ and } x \text{ is marked} \\
&\quad \text{where } \tau(x) = t[n], \text{ for some } n, \\
&\qquad \mathcal{M}' = \mathcal{M}[S(x) \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]], \\
&\qquad \mathcal{C}' = \mathbf{incvars}(z, S, \mathcal{C}) \\
&\qquad (\mathcal{M}'', \mathcal{C}'') = \mathbf{decref}(t)(\mathcal{M}(S(x))[S(y)], (\mathcal{M}', \mathcal{C}')) \\
(x[y \mapsto z], S, (\mathcal{M}, \mathcal{C})) &\to^{\#} (r, S'', (\mathcal{M}', \mathcal{C}''')), \text{otherwise} \\
&\quad \text{where } r = \mathbf{new}(\mathcal{M}), \\
&\quad \mathcal{M}' = \mathcal{M}[r \mapsto \mathcal{M}(S(x))[S(y) \mapsto S(z)]], \\
&\quad (\mathcal{C}', S') = \begin{cases} (\mathbf{decr}(S(z), \mathcal{C}), S[z \mapsto \texttt{nil}]) \\ \quad \text{if } z \text{ is marked} \\ (\mathcal{C}, S), \text{ otherwise,} \end{cases} \\
&\quad (\mathcal{C}'', S'') = \begin{cases} (\mathbf{decr}(S(x), \mathcal{C}', S[x \mapsto \texttt{nil}])) \\ \quad \text{if } x \text{ is marked} \\ (\mathcal{C}', S'), \text{ otherwise} \end{cases} \\
&\quad \mathcal{C}''' = (\#(\mathcal{M}'(r)) + \mathcal{C}'')[r \mapsto 1], \\
(\texttt{release}(\underline{x}, e), S, (\mathcal{M}, \mathcal{C})) &\to^{\#} (e, S[x \mapsto \texttt{nil}], \mathbf{decref}(\tau(x))(S(x), (\mathcal{M}, \mathcal{C}))) \\
&\quad \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})), \text{otherwise}
\end{aligned}
$$

Fig. 4: Operational Semantics of some reductions in RL

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 5.

$$\left(\begin{array}{l}\texttt{let } z = +(y,1) \texttt{ in } \texttt{swap}(\underline{x},\underline{y},\underline{z}), (y \mapsto 0, x \mapsto r),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{1}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\texttt{let } z = 1 \texttt{ in } \texttt{swap}(\underline{x},\underline{y},\underline{z}), (y \mapsto 0, x \mapsto r),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{2}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\texttt{pop}(\texttt{swap}(\underline{x},\underline{y},\underline{z})), (z \mapsto 1, y \mapsto 0, x \mapsto r),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{3}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } a = u[i] \texttt{ in } \ldots,\\ (j \mapsto 1, i \mapsto 0, u \mapsto r, \ldots, x \mapsto \texttt{nil}),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{4}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } a = 0 \texttt{ in } \ldots, (j \mapsto 1, i \mapsto 0, u \mapsto r, \ldots),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{5}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } b = u[j] \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{6}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } b = 1 \texttt{ in } \ldots, (a \mapsto 0, j \mapsto 1, \ldots),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{7}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } u' = \underline{u}[\underline{i} \mapsto \underline{b}] \texttt{ in } \ldots, (b \mapsto 1, a \mapsto 0, \ldots),\\ (r \mapsto \langle 0,1\rangle), (r \mapsto 2)\end{array}\right) \tag{8}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\ldots \texttt{let } u' = r' \texttt{ in } \ldots,\\ (b \mapsto 1, a \mapsto 0, \ldots, u \mapsto \texttt{nil}, \ldots),\\ (r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle), (r' \mapsto 1, r \mapsto 1)\end{array}\right) \tag{9}$$

$$\longrightarrow^{\#} (\ldots \underline{u'}[\underline{j} \mapsto \underline{a}] \ldots, (u' \mapsto r', b \mapsto 1, \ldots), (r' \mapsto \langle 1,1\rangle, \ldots)) \tag{10}$$

$$\longrightarrow^{\#} \left(\begin{array}{l}\texttt{pop}^7(r'), (u' \mapsto \texttt{nil}, b \mapsto 1, \ldots),\\ (r' \mapsto \langle 1,0\rangle, \ldots), (r' \mapsto 1, \ldots)\end{array}\right) \tag{11}$$

$$\overset{\star}{\longrightarrow}^{\#} \left(\begin{array}{l}r', (y \mapsto 0, x \mapsto \texttt{nil}),\\ (r' \mapsto \langle 1,0\rangle, r \mapsto \langle 0,1\rangle), (r' \mapsto 1, r \mapsto 1)\end{array}\right) \tag{12}$$

Fig. 5: An example reduction in *RL*

Notice how even though the reference count of r was 2 initially, we still saved a copy compared to Figure 3 and performed 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 4, it is an invariant that the count is accurate, that is, equation 13 holds, and the other invariants (**early-release**, **correct-marking**, and **release-marked**), are preserved as well.*

To establish a bisimulation between the *RL* state $(e', S', (\mathcal{M}', \mathcal{C}'))$ and the *FL* one $(e, S, \mathcal{M})$, we say these two states match if there exists a translation

function ρ 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 ρ to each of the references) when unmarking all the variables and removing all `release` constructors in e',
- For each variable x in $vars(e')$, $S(x) = \rho(S'(x))$,
- For each reference r in the domain of $\mathcal{M}'$ with a count greater than zero, $\mathcal{M}(\rho(r))$ is the result of translating the references of $\mathcal{M}'(r)$.

The mapping ρ is not fixed across the two executions but depends on the pairs of states being matched. For example, the bisimulation between the *FL* execution in Figure 3 and the *RL* execution in Figure 5 lines up the twelve states in each execution exactly. The mapping $\{r \mapsto r\}$ is used for matching states 1 through 8, and $\{r \mapsto r, r' \mapsto r'\}$ for states 9 and 10, and $\{r \mapsto r, r' \mapsto r''\}$ for states 11 and 12. Note that although it is not required for the mapping between references to be injective, it happens to be an invariant (that is not needed for proving the bisimulation result). We also do not need to assume that the store does not contain any cycles, though this too is an invariant that is preserved by both *FL* and *RL* executions.

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.*

As a step toward an imperative translation of *RL*, we extend *RL* by adding a construct of the form $\texttt{return}(e)$ to mark the return from a function call. Since the `return` label will be used as a variable in the imperative translation, it cannot be used as a variable identifier in *RL*. In the expanded language, $\texttt{return}(\square)$ is an evaluation context. We also have a redex of the form

$$(\texttt{return}(v), S, (\mathcal{M}, \mathcal{C})) \rightarrow^{\#} (v, S, (\mathcal{M}, \mathcal{C})).$$

The evaluation rule for function calls is modified as below.

$$\begin{aligned}(f(\overline{x}), S, (\mathcal{M}, \mathcal{C})) \rightarrow^{\#} (\texttt{return}(\texttt{pop}^n(e)), S_n, (\mathcal{M}, \mathcal{C}')), \text{ where}\\ \mathcal{C}' = \mathbf{incvars}(\overline{x}, S, \mathcal{C})\\ S_0 = \mathbf{push}(\overline{y} \mapsto S(\overline{x}), S)\\ S_{i+1} = \begin{cases} S_i[x_i \mapsto \texttt{nil}], \\ \quad \text{if } x_i \text{ is a marked and } S(x) \text{ is a reference} \\ S_i, \text{ otherwise} \end{cases}\end{aligned}$$

We also label the `pop` operations with the variable being popped so that it has the form $\texttt{pop}_x$ when the stack entry to be popped binds x. Neither of these extensions affects any of the claims about *RL* since the `return` operation essentially functions as a skip operation, and labeling the occurrences of `pop` has no impact on the evaluation. Both these constructs together with let-binding are used to define the stack employed in the imperative evaluation in terms of the *RL* stack.

The operation $\mathbf{stack}(x)(E)$ on an evaluation context E collects the stack of variables introduced by `return`, `pop`, and pending let-bindings on the path to the hole in the *RL* expression being evaluated. This operation fuses consecutive return variables so that the return value from the evaluation of a function is passed directly to the outermost return point. The stack used in the imperative evaluation binds variables in a somewhat different order than the operational semantics for *RL*. The **stack** operation is used to capture the sequence of variables that appear at the top of the stack during the evaluation of the imperative counterpart of an *RL* expression. It is used in defining the bisimulation between *RL* executions and the imperative semantics presented in the next section.

$$
\begin{aligned}
\mathbf{stack}(x)(\Box, st) &= st \\
\mathbf{stack}(x)(\texttt{let } y = a \texttt{ in } b, st) &= \mathbf{stack}(y)(a, \mathbf{push}(y, st)) \\
\mathbf{stack}(x)(\texttt{pop}_y(a), st) &= \mathbf{stack}(x)(a, \mathbf{push}(y, st)) \\
\mathbf{stack}(x)(\texttt{return}(a), st) &= \begin{cases} \mathbf{stack}(x)(a, st), & \text{if } x = \texttt{return} \\ \mathbf{stack}(\texttt{return})(a), \mathbf{push}(\texttt{return}, st)), & \text{otherwise} \end{cases}
\end{aligned}
$$

4 A Small Imperative Language

Mapping *RL* expressions to imperative code poses significant semantic challenges. In RL, we evaluate expressions, whereas in an imperative language, the statements are executed sequentially. RL evaluation returns a value, whereas the execution in an imperative language returns a state mapping variables to values. This changes the signalling mechanism used to identify the redexes. In *FL* and *RL*, a let-redex is triggered when the binding expression becomes a value. There is no such signalling mechanism in an imperative program since statements are executed successively. There is no handy equivalent of let-expressions $\texttt{let } x = a \texttt{ in } b$ in imperative languages since this expression is mapped to two statement blocks: s_a for computing a and assigning the value to x, and s_b for evaluating b. Since x is assigned in s_a and used in s_b, it has to be declared ahead of the block s_a, but this has the unfortunate side-effect of including s_a in its scope. Such issues of scope can be handled in a formalization based on the de Bruijn representation (as we do in our mechanized proofs), but require some care when using a named representation.

The target of our code generation is an imperative language *KL* which looks quite similar to *RL*. From *KL*, we can target a lower-level imperative language such as C that does not keep track of reference counts automatically. A program *KL* is defined as a sequence of functions, whose body is a statement, with the definitions in Figure 6.

$$
\begin{aligned}
e ::= {} & | \; n \\
& | \; x \\
& | \; \texttt{nil} \\
& | \; f(x_1, \ldots, x_n) \\
& | \; x[y] \\
& | \; x[y \mapsto z] \\
& | \; \texttt{newint}(n) \\
& | \; \texttt{newref}(n)
\end{aligned}
\qquad
\begin{aligned}
s ::= {} & | \; x := e \\
& | \; \texttt{ifnz } x \texttt{ then } s_1 \texttt{ else } s_2 \\
& | \; \texttt{skip} \\
& | \; s_1; s_2 \\
& | \; \{t \; x; s\} \\
& | \; \texttt{release } x \\
& | \; \texttt{pop}
\end{aligned}
$$

$$
\begin{aligned}
\text{decl} &::= t \; x \\
\text{function} &::= (\text{name}, \text{decl}^*, s) \\
\text{program} &::= \text{function}^*
\end{aligned}
$$

Fig. 6: Syntax of *KL*

As in *KL*, 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` within a program since it is there only for evaluation purposes. There is a special variable named `return` that is used as the return value of a function and is never used as a regular variable in a program.

Once a program with definitions Π is fixed, the evaluation state for *KL* is a triplet $(\mathcal{K}, S, (\mathcal{M}, \mathcal{C}))$, 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. $\mathcal{K}$ is the (possibly empty) statement (or continuation) being evaluated. The evaluation rules for *KL* are given in Figure 9 (for non-assignment statements), Figure 10 (for non-array assignment statements), and Figure 11 (for assignment statements).

Next, we illustrate the translation of *RL* expressions into *KL* code. To translate a function with body e from *RL* to *KL*, we use **translate**$(e, \texttt{return})$, where **translate** is defined in Figure 7. The operation assumes that in any *RL* subexpression of the form `let` $x = a$ `in` b, the variable x does not occur free in a.

To translate the *RL* program $\Delta^{\#}$ into a *KL* program Π, we translate each function definition $f(\overline{x}) = e$ in $\Delta^{\#}$ as $f(x_1, \ldots, x_d) = s_f$, where

$$s_f = \textbf{translate}(\texttt{pop}^d(e), \texttt{return}).$$

For the example of the $swap(u, i, j)$ program, the body

$$\texttt{let } a = u[i] \texttt{ in let } b = u[j] \texttt{ in let } u' = \underline{u}[\underline{i} \mapsto \underline{b}] \texttt{ in } \underline{u'}[\underline{j} \mapsto \underline{a}]$$

is translated as

$$\{\texttt{int } a; a := u[i]; \{\texttt{int } b; b := u[j]; \{\texttt{int}[2] \; u'; u' := \underline{u}[\underline{i} \mapsto \underline{b}]; \texttt{return} := \underline{u'}[\underline{j} \mapsto \underline{a}]\}\}\}$$

$$
\begin{aligned}
\textbf{translate}(n, x) &= x := n \\
\textbf{translate}(y, x) &= x := y \\
\textbf{translate}(\texttt{nil}, x) &= x := \texttt{nil} \\
\textbf{translate}(f(x_1, \ldots, x_n), x) &= x := f(x_1, \ldots, x_n) \\
\textbf{translate}(\texttt{let } (y : t) = e_1 \texttt{ in } e_2, x) &= \{t\ y; \textbf{translate}(e_1, y); \textbf{translate}(e_2, x)\} \\
\textbf{translate}(\texttt{ifnz } y \texttt{ then } e_1 \texttt{ else } e_2, x) &= \texttt{ifnz } y \texttt{ then } s_1 \texttt{ else } s_2, \\
&\quad\ s_1 = \textbf{translate}(e_1, x), \\
&\quad\ s_2 = \textbf{translate}(e_2, x) \\
\textbf{translate}(y[z], x) &= x := y[z] \\
\textbf{translate}(y[z \mapsto w], x) &= x := y[z \mapsto w] \\
\textbf{translate}(\texttt{newint}(n), x) &= x := \texttt{newint}(n) \\
\textbf{translate}(\texttt{newref}(n), x) &= x := \texttt{newref}(n) \\
\textbf{translate}(\texttt{release}(y, e), x) &= \texttt{release } y; \textbf{translate}(e, x) \\
\textbf{translate}(\texttt{pop}_y(e), x) &= \textbf{translate}(e, x); \texttt{pop}_y \\
\textbf{translate}(\texttt{return}(e), x) &= \begin{cases} \textbf{translate}(e, x), \text{ if } x = \texttt{return} \\ \{t\ \texttt{return}; s_e; x := \texttt{return}\}, \text{ otherwise,} \\ \quad \textit{where} \\ \quad s_e = \textbf{translate}(e, \texttt{return}), \\ \quad t = \tau(e) \end{cases}
\end{aligned}
$$

Fig. 7: Translation from *RL* to *KL*

Next, we demonstrate a bisimulation between the evaluation of an *RL* expression and the execution of its translated program.

We define $\textbf{lvars}(S)$ to represent the stack of variables bound in the stack.

$$
\begin{aligned}
\textbf{lvars}(\textbf{push}(x \mapsto v, S)) &= \textbf{push}(x, \textbf{lvars}(S)) \\
\textbf{lvars}(\textbf{push}(x \mapsto \texttt{undef}, S)) &= \textbf{lvars}(S) \\
\textbf{lvars}(\textit{empty}) &= \textit{empty}
\end{aligned}
$$

The operation $\textbf{defined}(S)$ extracts the defined bindings in the stack S:

$$
\begin{aligned}
\textbf{defined}(\textbf{push}(x \mapsto v, S)) &= \textbf{push}(x \mapsto v, \textbf{lvars}(S)) \\
\textbf{defined}(\textbf{push}(\texttt{return} \mapsto v, S)) &= \textbf{defined}(S) \\
\textbf{defined}(\textbf{push}(x \mapsto \texttt{undef}, S)) &= \textbf{defined}(S) \\
\textbf{defined}(\textit{empty}) &= \textit{empty}
\end{aligned}
$$

$$
\begin{pmatrix}
\{\texttt{int}\ z; z := +(y,1); \texttt{return} := \texttt{swap}(\underline{x},\underline{y},\underline{z})\}; \mathcal{K}, \\
(\texttt{return} \mapsto \texttt{undef}, y \mapsto 0, x \mapsto r), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{1}
$$

$$
\longrightarrow^{!} \begin{pmatrix}
z := +(y,1); \texttt{return} := \texttt{swap}(\underline{x},\underline{y},\underline{z}); \texttt{pop}; \mathcal{K}, \\
(z \mapsto \texttt{undef}; \texttt{return} \mapsto \texttt{undef}, y \mapsto 0, x \mapsto r), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{2}
$$

$$
\longrightarrow^{!} \begin{pmatrix}
\texttt{return} := \texttt{swap}(\underline{x},\underline{y},\underline{z}); \texttt{pop}; \mathcal{K}, \\
(z \mapsto 1, \texttt{return} \mapsto \texttt{undef}, y \mapsto 0, x \mapsto r), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{3}
$$

$$
\longrightarrow^{!} \begin{pmatrix}
s_{swap}; \texttt{pop};^{3} \texttt{pop}; \mathcal{K}, \\
(j \mapsto 1, i \mapsto 0, u \mapsto r, z \mapsto 1, \texttt{return} \mapsto \texttt{undef}, y \mapsto 0, x \mapsto \texttt{nil}), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{4}
$$

$$
\longrightarrow^{!} \begin{pmatrix}
a := u[i]; \ldots; \texttt{pop};^{5} \mathcal{K}, \\
(a \mapsto \texttt{undef}, j \mapsto 1, i \mapsto 0, u \mapsto r, z \mapsto 1, \texttt{return} \mapsto \texttt{undef}, \ldots), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{5}
$$

$$
\xrightarrow{\star}{}^{!} \begin{pmatrix}
b := u[j]; \ldots; \texttt{pop};^{6} \mathcal{K}, \\
(b \mapsto \texttt{undef}, a \mapsto 0, j \mapsto 1, \ldots), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{6}
$$

$$
\xrightarrow{\star}{}^{!} \begin{pmatrix}
u' := \underline{u}[\underline{i} \mapsto \underline{b}]; \ldots; \ldots; \texttt{pop};^{7} \mathcal{K}, \\
(u' \mapsto \texttt{undef}, b \mapsto 1, a \mapsto 0, j \mapsto 1, \ldots), \\
((r \mapsto \langle 0,1\rangle), (r \mapsto 2))
\end{pmatrix} \tag{7}
$$

$$
\xrightarrow{\star}{}^{!} \begin{pmatrix}
\texttt{return} := \underline{u'}[\underline{j} \mapsto \underline{a}]; \ldots; \mathcal{K}, \\
(u' \mapsto r; b \mapsto 1, a \mapsto 0, u \mapsto \texttt{nil}, \ldots), \\
((r' \mapsto \langle 1,1\rangle, r \mapsto \langle 0,1\rangle), (r' \mapsto 1, r \mapsto 1))
\end{pmatrix} \tag{8}
$$

$$
\xrightarrow{\star}{}^{!} (\mathcal{K}, (\texttt{return} \mapsto r', \ldots), ((r' \mapsto \langle 1,0\rangle, \ldots), (r' \mapsto 1, r \mapsto 1))) \tag{9}
$$

Fig. 8: An example reduction in *KL*

For *KL* program s, let $\mathbf{body}(s)$ be defined as below.

$$
\begin{aligned}
\mathbf{body}(\{t\ x; s\}) &= \mathbf{body}(s); \texttt{pop} \\
\mathbf{body}(s_1; s_2) &= \mathbf{body}(s_1); s_2
\end{aligned}
$$

A rough point of correspondence between an *RL* state $(e, S_1, (\mathcal{M}, \mathcal{C}))$ and *KL* state $(s_e, S_2, (\mathcal{M}_2, \mathcal{C}_2))$ is:

1. The *KL* program s_e corresponding to e is the empty program when e is a value, and is $\mathbf{body}(\mathbf{translate}(e, \texttt{return}))$, otherwise.
2. The correspondence between the *RL* stack S_1 and *KL* stack S_2 is that $\mathbf{lvars}(S_2) = \mathbf{stack}(\texttt{return})(e, \mathbf{push}(\texttt{return}, \mathit{empty})) \circ \mathbf{lvars}(S)$ for some S,

$$
\begin{aligned}
(\texttt{skip}; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^{!} (\mathcal{K}, S, (\mathcal{M}, \mathcal{C})) \\
(\{t\ x; s\}; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^{!} \begin{pmatrix} s; \texttt{pop}; \mathcal{K}, \mathbf{push}(x \mapsto \texttt{undef}, S) \\ (\mathcal{M}, \mathcal{C}) \end{pmatrix} \\
(\texttt{ifnz}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^{!} (s_i; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})),\ \text{where} \\
& \qquad i = \begin{cases} 2, & \text{if } S(x) = 0 \\ 1, & \text{otherwise} \end{cases} \\
(\texttt{release}\ x; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^{!} (\mathcal{K}, S[x \mapsto \texttt{nil}], (\mathcal{M}', \mathcal{C}')),\ \text{where} \\
& \qquad (\mathcal{M}', \mathcal{C}') = \mathbf{decref}(S(x), (\mathcal{M}, \mathcal{C})) \\
(\texttt{pop}; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^{!} (\mathcal{K}, \mathbf{pop}(S), (\mathcal{M}, \mathcal{C}))
\end{aligned}
$$

Fig. 9: Operational Semantics of *KL*: Non-assignment statements

and $\mathbf{defined}(S_2) = \mathbf{push}(\texttt{return} \mapsto v, S_1)$, when e is the value v, and otherwise $S_1 = \mathbf{defined}(S_2)$.

3. $(\mathcal{M}_1, \mathcal{C}_1) = (\mathcal{M}_2, \mathcal{C}_2)$.

The idea is that we initiate both evaluations with a stack S_0 but the *KL* state stack is of the form $\mathbf{push}(\texttt{return} \mapsto \texttt{undef}, S_0)$ to capture the return value. Furthermore, the S_2 stack contains bindings, defined and undefined, for the variables in $\mathbf{stack}(\texttt{return})(e)$, whereas all the bindings in S_1 are defined.

However, the first bullet holds only for canonical states, as defined below. For example, in *RL*, e can have the form $\texttt{let}\ x = r\ \texttt{in}\ e'$ for some reference r but the syntax of *KL* does not allow explicit references in expressions. The fourth state in Figure 8 has a program of the form $\{\texttt{int}\ a; a := u[i]; \ldots\}$, which is not the body of the program. To get around these discrepancies, we restrict the correspondence to states in canonical form obtained by applying certain reductions. In *RL*, in any state $(e, S_1, (\mathcal{M}_1, \mathcal{C}_2))$, redexes e' of the form $\texttt{let}\ x = v\ \texttt{in}\ e''$, $\texttt{return}(v)$, and $\texttt{pop}(v)$, where $e = E[e']$, must be silently reduced. Similarly, in any *KL* state $(s; S_2, (\mathcal{M}_2, \mathcal{C}_2))$ any of the following redexes must be silently reduced:

1. $\{t\ x; s\}; \mathcal{K}$
2. $x := v; \mathcal{K}$
3. $x := \texttt{return}; \mathcal{K}$
4. $\texttt{pop}; \mathcal{K}$

With these reductions, the above correspondence yields a bisimulation between *RL* and *KL* execution steps. For example, in the derivation in Figure 8 (assuming that continuation $\mathcal{K}$ is empty) the canonical states are 2, 3, 5, 6, 7, 8, and 9, which correspond to the *RL* evaluation states 1, 3, 4, 6, 8, 10, and 12 in Figure 5. The correspondence between the canonical *RL* expressions in evaluation in Figure 5 and their translations in the *KL* evaluation in Figure 8 is shown in Figure 12.

$$
\begin{aligned}
(x := \underline{y}; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^! (\mathcal{K}, S[x \mapsto S(y)], (\mathcal{M}, \mathcal{C})) \\
(x := y; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^! (\mathcal{K}, S[x \mapsto S(y)], (\mathcal{M}, \mathcal{C}')), \\
&\quad \text{where } y \text{ is unmarked}, \\
&\quad \mathcal{C}' = \mathbf{incr}(S(y), \mathcal{C}) \\
(x := n; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) &\longrightarrow^! (\mathcal{K}, S[x \mapsto n], (\mathcal{M}, \mathcal{C})) \\
(y := f(x_1, \ldots, x_n); \mathcal{K}, S, (\mathcal{M}, \mathcal{C}')) &\longrightarrow^! (\mathcal{K}', S_n, (\mathcal{M}, \mathcal{C}')), \text{ where} \\
&\quad \mathcal{K}' = \begin{cases} s_f; \mathtt{pop}^n; \mathcal{K}, & \text{if } y = \mathtt{return} \\ s_f; \mathtt{pop};^n y := \underline{\mathtt{return}}; \mathtt{pop}; \mathcal{K}, & \\ \text{otherwise} & \end{cases} \\
&\quad S' = \begin{cases} \mathbf{push}(\overline{y} \mapsto S(\overline{x}), & \\ \qquad \mathbf{push}(\mathtt{return} \mapsto \mathtt{undef}, S)), & \\ \quad \text{if } y \neq \mathtt{return} & \\ \mathbf{push}(\overline{y} \mapsto S(\overline{x}), S), \text{ otherwise} & \end{cases} \\
&\quad S_0 = S' \\
&\quad S_{i+1} = \begin{cases} S_i[x_i \mapsto \mathtt{nil}], \text{ if} & \\ \quad x_i \text{ is a marked and} & \\ \quad S(x) \text{ is a reference} & \\ S_i, \text{ otherwise} & \end{cases} \\
&\quad \mathcal{C}' = \mathbf{incvars}(\overline{x}, S, \mathcal{C})
\end{aligned}
$$

Fig. 10: Operational Semantics of *KL*: Non-Array Assignment Statements

$\mathtt{let}\ z = +(y, 1)\ \mathtt{in}\ \mathtt{swap}(\underline{x}, \underline{y}, \underline{z})$	$z := +(y, 1); \mathtt{return} := \mathtt{swap}(\underline{x}, \underline{y}, \underline{z}); \mathtt{pop};$
$\mathtt{swap}(\underline{x}, \underline{y}, \underline{z})$	$\mathtt{return} := \mathtt{swap}(\underline{x}, \underline{y}, \underline{z}); \mathtt{pop};$
$\mathtt{return}(\mathtt{let}\ a = u[i]\ \mathtt{in}\ \ldots)$	$a := u[i]; \ldots; \mathtt{pop};^5$
$\mathtt{return}(\mathtt{let}\ b = u[j]\ \mathtt{in}\ \ldots)$	$b := u[j]; \ldots; \mathtt{pop};^6$
$\mathtt{return}(\mathtt{let}\ u' = \underline{u}[\underline{i} \mapsto \underline{b}]\ \mathtt{in}\ \ldots)$	$u' := \underline{u}[\underline{i} \mapsto \underline{b}]; \ldots; \mathtt{pop};^7$
$\mathtt{return}(\underline{u'}[\underline{j} \mapsto \underline{a}])$	$\mathtt{return} := \underline{u'}[\underline{j} \mapsto \underline{a}]; \ldots; \mathtt{pop};^7$
r'	

Fig. 12: Correspondence between *RL* and *KL* Evaluation

Theorem 5. *The reduction relations* $\longrightarrow^{\#}$ *in* RL *and* $\longrightarrow^!$ *in KL are in bisimulation.*

5 Conclusions

Functional languages offer significant advantages in terms of expressiveness and verifiability, but they require fairly extensive runtime support. Our goal here

$$
\begin{aligned}
(x := y[z]; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) \longrightarrow^! & \ (\mathcal{K}, S[x \mapsto v], (\mathcal{M}, \mathcal{C}')), \text{ where} \\
& v = \mathcal{M}(S(y))[S(z)], \mathcal{C}' = \mathbf{incr}(v, \mathcal{C}) \\
(x := \underline{y}[z \mapsto w]; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) \longrightarrow^! & \ (\mathcal{K}, S[x \mapsto v], (\mathcal{M}'', \mathcal{C}'')), \text{ where} \\
& v = S(y), \mathcal{C}(v) = 1, v \text{ is a reference}, \\
& \mathcal{M}' = \mathcal{M}(v)[S(z) := S(w)], \\
& \mathcal{C}' = \mathbf{incvars}(w, S, \mathcal{C}), \\
& (\mathcal{M}'', \mathcal{C}'') = \mathbf{decref}(t)(\mathcal{M}(v)[S(z)], (\mathcal{M}', \mathcal{C}')) \\
(x := y[z \mapsto w]; \mathcal{K}, S, (\mathcal{M}, \mathcal{C})) \longrightarrow^! & \ (\mathcal{K}, S[x \mapsto v], (\mathcal{M}', \mathcal{C}'')), \text{ where} \\
& v = S(y), v \text{ is a reference}, \\
& y \text{ is unmarked or } \mathcal{C}(v) > 1, \\
& r = \mathbf{new}(\mathcal{M}), \\
& \mathcal{M}' = \mathcal{M}[r \mapsto \mathcal{M}(S(y))[S(z) := S(w)], \\
& \mathcal{C}' = (\#\mathcal{M}'(r) + \mathcal{C})[r \mapsto 1], \\
& \mathcal{C}'' = \mathbf{decr}(\mathcal{M}(v)(S(z)), \mathcal{C}')
\end{aligned}
$$

Fig. 11: Operational Semantics of *KL*: Assignment statements

is to generate efficient, standalone code from an executable fragment of a logic in which we can unify specification, modeling, and execution. The PVS2C code generator translates an applicative fragment of PVS into C code. The generated C code is self-contained and does not rely on a run time. The generated code preserves the type safety of the typechecked PVS. It can only crash by exhausting resource bounds. The generated C code is comparable in efficiency to the corresponding hand-crafted C, and is typically a lot faster than the Common Lisp code generated from PVS.

The intermediate languages presented here: *FL*, *RL*, and *KL* form the core of the translation from the applicative subset of PVS to executable C code. The translations between these languages and the bisimulation proofs presented here form a step toward the mechanized verification of the code generator. We believe that the proof outlined in the paper can be easily mechanized, and can also be used as a foundation for similar proofs involving more sophisticated language features.

Acknowledgment. This work was supported by the National Institute of Aerospace Award C18-201097-SRI, NSF Grant SHF-1817204, and DARPA under agreement number HR001119C0075. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NASA, NSF, DARPA, or the U.S. Government. We thank the anonymous referees for their constructive feedback.

References

1. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.
2. David Aspinall and Martin Hofmann. Another type system for in-place update. In *European Symposium on Programming (ESOP) , Grenoble, France*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, April 2002.
3. William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
4. Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996.
5. George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.
6. Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. Opal: Design and implementation of an algebraic programming language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures: International Conference Zurich, Switzerland, March 2–4,1994 Proceedings*, pages 228–244, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
7. M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, September 1993.
8. Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In S. Kowalewshi and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2009.
9. 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.
10. Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In Sanjai Rayadurgam and Oksana 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.
11. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 502–514. ACM, 1993.
12. K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *16th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1989.
13. P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, pages 351–363. ACM, August 1986.
14. Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 300–314, New York, NY, USA, 1985. ACM.

15. A. Kanade, A. Sanyal, and U. Khedker. A PVS based framework for validating compiler optimizations. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*, 2006.
16. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.
17. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
18. J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.
19. Luc Moreau and Jean Duprat. A construction of distributed reference counting. *Acta Inf*, 37(8):563–595, 2001.
20. W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.
21. Wolfram Schulte. Deriving reference count garbage collectors. In *6th International Symposium on Programming Language Implementation and Logic Programming*, pages 102–116, September 1994.
22. Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In A. Pettorossi, editor, *11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 01)*, Lecture Notes in Computer Science, pages 1–24. Springer-Verlag, 2002.
23. Natarajan Shankar. A brief introduction to the PVS2C code generator. In Bruno Dutertre and Natarajan Shankar, editors, *AFM@NFM*, volume 5 of *Kalpa Publications in Computing*, pages 109–116. EasyChair, 2017.
24. David W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness.* PhD thesis, University of York, Department of Computer Science, York, England, March 1998.
25. Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. *CoRR*, abs/1908.05647, 2019. Appears in pre-conference proceedings of IFL'19: `http://2019.iflconference.org/pre-conference-proceedings.pdf`.
26. Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of the IEEE Conference on Computer Languages '98*, pages 184–193. IEEE, April 1998.