Kalpa Publications in Computing

Volume 5, 2018, Pages 109–116

Automated Formal Methods

A Brief Introduction to the PVS2C Code Generator [*]

Natarajan Shankar

SRI International Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025 USA
shankar@csl.sri.com

Abstract

We present a brief tutorial on the PVS2C code generator for producing C code from an applicative fragment of the PVS specification language. This fragment roughly corresponds to a self-contained functional language. The tutorial covers the generation of C code for numeric data types and associated operations, arrays, recursive data types, and higher-order operations.

1 Introduction

Specification languages are meant to capture the “what” of computation while programming language express the “how”. For this reason, a specification language need not be executable. However, many specification languages do contain executable sublanguages. Execution is useful for validating specifications, generating verified software and systems, and for performing large calculations within proofs. Code generation makes it possible to construct executable systems without having to formalize programming notations and their semantics within the specification language, or building special-purpose verification tools that target these programming languages.

The Prototype Verification System (PVS) is an interactive proof assistant with an expressive specification language based on higher-order logic. The type system admits predicate subtypes, dependent tuple, record, and function types, and recursive datatypes. The language also supports parametric theories. The expression language includes function application, lambda abstraction, quantification, conditional expressions, LET-binding, and record/tuple/function updates. The quantifier-free fragment of the language can be viewed as an applicative language.

There are two basic problems in mapping an applicative language to an imperative one. The first is that applicative semantics require copying on updates. This kind of copying can be

[*] This work was supported by NSF Grant CSR-EHCS(CPS)-0834810, NASA Cooperative Agreement NNA10DE73C, and by DARPA under agreement number FA8750-12-C-0284 and FA8750-16-C-0043. 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 NSF, NASA, DARPA or the U.S. Government.

very expensive: sorting a 1000-element array can involve thousands of copies. It is therefore important to identify and exploit opportunities for in-place updates. The second problem is of course the execution of an applicative program can generate memory that is no longer referenced and needs to be garbage-collected. In PVS2C [1], reference counting is used to address both issues. In simple terms, an array can be updated in place when its reference count is one, and it can be garbage collected when its reference count drops to zero. Since reference cycles cannot be created when executing PVS, reference counting does ensure that no live references are collected and all dead references are garbage-collected. More strongly, references are released as soon as possible so as to maximize the opportunities for in-place updates. The execution of well-typed PVS expressions is safe: the only possible runtime errors are when the execution exhausts heap or stack space. Typechecking, particularly through the discharging of type correctness condition (TCC) proof obligations, ensures that there can be no buffer overflows, null dereferences, uncaught exceptions, division by zero, etc.

Though PVS2C targets the C programming language, the translation from PVS to C is factored through an intermediate representation (IR) that can be used to target other programming languages. The IR is based on A-normal form [2]. The `pvs2ir` operation translates PVS expressions into the IR. This translation basically involves flattening expressions to create variable bindings for subexpressions. The IR includes some type information to help track array sizes. The `ir2c` operation maps IR expressions into C by essentially converting the LET-bindings into assignments. The PVS2C generator can be invoked as `M-x pvs-c-theory` with the cursor on a theory in a `.pvs` file. The code generator generates a header and code file for the given theory as well as for any theories that are in the import chain. The code generator currently handles Boolean, numeric, record, tuple, recursive datatypes, and function types. Fixed width, uni-dimensional arrays are handled using C arrays, and the others are treated as function types. We are working on extending the translation to dependently sized array types and polymorphic types. We present a short tutorial on the use of the prototype implementation of PVS2C.

2 A Small Example

We first present a small example to illustrate the flow with the theory `smallswap` shown below. The type `nat32` is a subtype that captures the C type uint_32. The `pvs2ir` translator uses the Common Lisp ground evaluator [3] to evaluate such expressions. This also holds for the `numrows` parameter. The type `smallarray` represents an array of size `5`. The `swap` operation is defined to exchange `A(i)` with `A(j)`.

```
swap : THEORY
  BEGIN

  nat32: TYPE = below(exp2(32))

  numrows:  nat32 = 5

  rows: TYPE = below(numrows)

  smallArray: TYPE = [rows -> nat32]

  A: VAR smallArray

  swap(A, (i, j : rows)): smallArray =
    A WITH [(i) := A(j),
            (j) := A(i)]

  test: smallArray =
   (LET A  = (LAMBDA (i: rows): i)
     IN swap(A, 2, 3))

  END swap
```

This generates the IR shown below. The two lookups of `A(j)` and `A(i)` are bound to the variables `ivar_7` and `ivar_10`, respectively. The swap operation is made up of two updates. The variables are printed with their type information. Note that several of the variable occurrences are as arguments to the `last` operator. This operator marks the last occurrence of a variable in an evaluation path. It is used in the C translator for helping account for references. The whole definition is represented as a lambda expression where the body is given a return type following the arrow '`->`'.

```
(lambda ((ivar_1 swap_smallArray) (ivar_2 (subrange 0 4))
         (ivar_3 (subrange 0 4)))
  '->
  swap_smallArray
  (let ivar_4
    (subrange 0 4294967295)
    (let ivar_7
      (subrange 0 4294967295)
      (lookup (ivar_1 swap_smallArray) (ivar_3 (subrange 0 4)))
      (last (ivar_7 (subrange 0 4294967295))))
    (let ivar_5
      (subrange 0 4294967295)
      (let ivar_10
        (subrange 0 4294967295)
        (lookup (ivar_1 swap_smallArray) (ivar_2 (subrange 0 4)))
        (last (ivar_10 (subrange 0 4294967295))))
      (let ivar_17
        swap_smallArray
        (update (last (ivar_1 swap_smallArray))
         (last (ivar_2 (subrange 0 4)))
         (last (ivar_4 (subrange 0 4294967295))))
        (let ivar_22
          swap_smallArray
          (update (last (ivar_17 swap_smallArray))
           (last (ivar_3 (subrange 0 4)))
           (last (ivar_5 (subrange 0 4294967295))))
          (last (ivar_22 swap_smallArray)))))))
```

The operation `ir2c` generates the C counterpart of the IR translation. Two files: `swap_c.h` and `swap_c.c` are generated. The header file `swap_c.h` contains the following `include` declarations.

```
#include <stdio.h>

#include <stdlib.h>

#include <inttypes.h>

#include <stdbool.h>

#include <string.h>

#include <gmp.h>

#include "pvslib.h"

#include "exp2_c.h"
```

It also contains the type definition corresponding to the `smallArray` type. The array is defined by a `struct` that has a reference count field `count`, and the C array `elems`. We also define five operations for each such aggregate type: `new`, which constructs a fresh array; `release`, which decreases the reference count by one while freeing the struct if the reference count drops to zero; `copy`, which does a shallow copy; `equal`, which is a recursive equality test; and `update`, which performs an update.

```
struct swap_smallArray_s { uint32_t count;
 uint32_t elems[5]; };
typedef struct swap_smallArray_s * swap_smallArray_t;

extern swap_smallArray_t new_swap_smallArray(void);

extern void release_swap_smallArray(swap_smallArray_t x);

extern swap_smallArray_t copy_swap_smallArray(swap_smallArray_t x);

extern bool_t equal_swap_smallArray(swap_smallArray_t x, swap_smallArray_t y);

extern swap_smallArray_t
   update_swap_smallArray(swap_smallArray_t x, uint32_t i, uint32_t v);
```

The file `swap_c.c` contains the definitions of the above operations, as well as the definition of `swap`. Each sub-expression in the IR definition of `swap` is translated with a return variable, where each LET-binding turns into an assignment with a possible casting. It might seem surprising that there is no explicit reference counting in the definition. The update operation manages the reference count for the array being updated. Since the two update operations are applied to variables marked as `last`, they will be executed in place if the reference count of the array passed into the operation as `ivar_1` has a reference count of one.

```
extern swap_smallArray_t f_swap_swap(swap_smallArray_t ivar_1,
                                     uint8_t ivar_2,
                                     uint8_t ivar_3){
        swap_smallArray_t result;
        uint32_t ivar_4;
        uint32_t ivar_7;
        ivar_7 = (uint32_t)ivar_1->elems[ivar_3];
        ivar_4 = (uint32_t)ivar_7;
        uint32_t ivar_5;
        uint32_t ivar_10;
        ivar_10 = (uint32_t)ivar_1->elems[ivar_2];
        ivar_5 = (uint32_t)ivar_10;
        swap_smallArray_t ivar_17;
        ivar_17 = (swap_smallArray_t)
                  update_swap_smallArray(ivar_1, ivar_2, ivar_4);
        swap_smallArray_t ivar_22;
        ivar_22 = (swap_smallArray_t)
                  update_swap_smallArray(ivar_17, ivar_3, ivar_5);
        result = (swap_smallArray_t)ivar_22;

        return result;
}
```

The generated programs can be tested by means of a hand-written main such as the one shown below.

```
#include "swap_c.h"

int main(){
  swap_smallArray_t result;
  result = f_swap_test();
  printf("\n result->count = %u", result->count);
  printf("\n");
  for (uint32_t i = 0; i < f_swap_numrows(); i++){
    printf("a[%u] = %u; ", i, result->elems[i]);
  }
  printf("\n");
}
```

Compiling and executing this program generates.

```
 result->count = 1
a[0] = 0; a[1] = 1; a[2] = 3; a[3] = 2; a[4] = 4;
```

3 Arithmetic Operations

We next examine the translation of arithmetic operations. The theory `arithops` shows some of the types and a few variations on addition.

```
arithops: THEORY
  BEGIN
     uint8: TYPE = below(exp2(8))
     uint16: TYPE = upto(exp2(16) - 1)
     uint32: TYPE = upto(exp2(32) - 1)
     uint64: TYPE = upto(exp2(64) - 1)
     uint128: TYPE = upto(exp2(128) - 1)

     int8: TYPE = subrange(-exp2(7), exp2(7) - 1)
     int16: TYPE = subrange(-exp2(15), exp2(15) - 1)
     int32: TYPE = subrange(-exp2(31), exp2(31) - 1)
     int64: TYPE = subrange(-exp2(63), exp2(63) - 1)
     int128: TYPE = subrange(-exp2(127), exp2(127) - 1)
     addu8u8_u8: uint8 = 127 + 128
     addu8u8_u16: uint16 = 255 + 255
     addu8u16_u16: uint16 = 255 + 65000
     addu16u8_u16: uint16 = 65000 + 255
     addu16u16_u16: uint16 = 32768 + 32767
     addu16u16_u8: uint8 = (LET x : uint16  = 127,
                                  y : uint16 = 128
                                 IN x + y)
     addu16u16_u32: uint32 = 65535 + 65535
     addu16u32_u32: uint32 = 65535 + 4294900000
     addu32u32_u32: uint32 = 2094900000 + 2094900000
     addu32u32_u16: uint16 = (LET x : uint32  = 32000,
                                   y : uint32 = 32000
                                 IN x + y)
  END arithops
```

We show the generated code for the last operation `addu32u32_u16`. This code fragment illustrates the casting between the different numeric types. PVS2C handles casting between signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit numbers, as well as multi-precision representations.

```
extern uint16_t f_arithops_addu32u32_u16(void){
        uint16_t result;
        uint32_t ivar_1;
        ivar_1 = (uint32_t)32000;
        uint32_t ivar_2;
        ivar_2 = (uint32_t)32000;
        result = (uint16_t)(ivar_1 + ivar_2);
        return result;
}
```

The multi-precision computations use the Gnu Multi-Precision (GMP) library. The operations employ a different calling convention where the first argument to the operation is the variable used for recording the result.

```
extern void f_arithops_addu32u32_nat(mpz_t result){
        uint32_t ivar_1;
        ivar_1 = (uint32_t)4294967295;
        uint32_t ivar_2;
        ivar_2 = (uint32_t)4294967295;
        mpz_set_ui(result, (uint64_t)ivar_1);
        mpz_add_ui(result, result, (uint64_t)ivar_2);

}
```

PVS record datatypes are represented as C structs with the corresponding fields. The PVS declaration for the record type `smallPair` is shown below as consisting of three fields: `left`, `right`, and `mid`.

```
   smallPair: TYPE = [# left, right : smallArray, mid: nat32 #]
```

The corresponding C type is shown below.

```
struct smallswap_smallPair_s {
        uint32_t count;
        smallswap_smallArray_t left;
        uint32_t mid;
        smallswap_smallArray_t right;};
typedef struct smallswap_smallPair_s * smallswap_smallPair_t;
```

As with arrays, the record types also have five generated operations for creating a new object, copying an object, checking for equality, updating an object, and releasing the object.

```
extern smallswap_smallPair_t new_smallswap_smallPair(void);

extern void release_smallswap_smallPair(smallswap_smallPair_t x);

extern smallswap_smallPair_t copy_smallswap_smallPair(smallswap_smallPair_t x);

extern bool_t
   equal_smallswap_smallPair(smallswap_smallPair_t x, smallswap_smallPair_t y);

extern smallswap_smallPair_t
   update_smallswap_smallPair_left(smallswap_smallPair_t x,
                                   smallswap_smallArray_t v);

extern smallswap_smallPair_t
   update_smallswap_smallPair_mid(smallswap_smallPair_t x, uint32_t v);

extern smallswap_smallPair_t
   update_smallswap_smallPair_right(smallswap_smallPair_t x,
                                    smallswap_smallArray_t v);
```

PVS n-tuples are treated as record types with fields `project_1` to `project_n`.

Recursive datatypes in PVS are introduced with constructors along with their recognizers and accessors. The declaration for `numlist` introduces a datatype with two constructors: `nnull` and `ncons`, where `nnull` has no accessors and the recognizer `nnull?`, and `ncons` has two accessors: `ncar` and `ncdr`, and the recognizer `ncons?`.

```
    numlist: DATATYPE
     BEGIN
       nnull: nnull?
       ncons(ncar: nat32, ncdr: numlist): ncons?
     END numlist
```

The translation to C first generates a parent datatype with just the `count` field for the reference count and an `index` field for marking the index of the constructor.

```
struct drev_numlist_adt_s {
         uint32_t count;
        uint8_t drev_numlist_adt_index;};
typedef struct drev_numlist_adt_s * drev_numlist_adt_t;
```

For each nontrivial constructor, there is a struct extending the parent struct with the relevant fields. For example, the constructor `ncons` yields the struct definition below extending the struct `drev_numlist_adt_s`.

```
struct drev_ncons_s {
        uint32_t count;
        uint8_t drev_numlist_adt_index;
        uint32_t ncar;
        drev_numlist_adt_t ncdr;};
typedef struct drev_ncons_s * drev_ncons_t;
```

Datatype updates are handled in the same way as structs so that it is possible to define destructive counterparts for appending and reversing lists. PVS2C does not yet handle parametric datatypes nor parametric theories.

PVS2C handles closures by first generating a parent C struct for the function type with fields for

- The reference `count`
- The unary function pointer `fptr`
- The multiary function pointer `mptr`
- The release function pointer `rptr`
- The copy function pointer `cptr`, and
- A hashtable for storing function updates.

The corresponding C representation is defined below. The actual value representing a closure also contains a field for the bindings for the free variables.

```
struct closr_closure_0_s { uint32_t count;
        uint32_t (* fptr)(struct closr_closure_0_s *, uint32_t);
        uint32_t (* mptr)(struct closr_closure_0_s *, uint32_t);
        void (* rptr)(struct closr_closure_0_s *);
        struct closr_closure_0_s * (* cptr)(struct closr_closure_0_s *);
        closr_closure_0_htbl_t htbl;};
```

4 Conclusions

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 a lot faster than the Common Lisp code generated from PVS.

We are working on extending the translation to cover parametric theories, dependently-sized arrays, and strings. We also plan to integrate it with the random testing capability in order to test the generated code on large sets of test vectors. The intermediate language is independently useful and we plan to support it with various forms of static analysis that can improve the quality of the generated code. Eventually, we would also like to handle specifications of concurrent systems so that we can generate monitors and entire systems starting from abstract specifications.

Acknowledgments. This material is based on work supported by NASA NRA NNA13AC55C, NSF Grant CNS-0917375, and by DARPA and the United States Air Force under agreement contract number FA8750-16-C-004315-C-0010. 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, USAF, DARPA, or the U.S. Government. We thank the anonymous referees for their constructive feedback.

References

[1] 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.

[2] 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.

[3] 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. Available at ftp://ftp.csl.sri.com/pub/users/shankar/lopstr01.pdf.