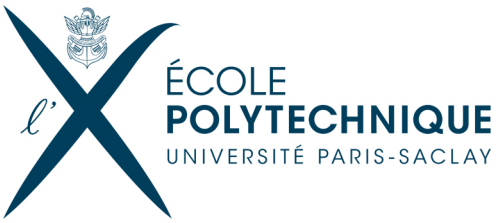
ÉCOLE POLYTECHNIQUE
PROMOTION X2011
Férey Gaspard



RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

Titre

NON CONFIDENTIEL

Option :	INFORMATIQUE
Champ de l'option :	Math-Informatique
Directeur de l'option :	Olivier Bournez
Directeur de stage :	Olivier Bournez
Dates du stage :	7 avril - 22 aout 2014
Nom et adresse de l'organisme :	SRI International Computer Science Laboratory (CSL) 333 Ravenswood Avenue Menlo Park, CA 94025-3493 United States

August 1, 2014

Abstract

SAT solvers can generate proofs that justify their final result. This report presents a verified program that checks if the generated proofs are valid. The checker is written and proved in PVS. In order to execute it, Lisp code is automatically generated. We describe a trusted kernel called the Kernel of Truth. Finally, we prove that it exists a valid kernel proof corresponding to any valid checker proof. This demonstrates that the checker is a safe extension of the kernel.

Contents

1 Introduction

Before starting, I would like to thank Dr. Natarajan Shankar for making this internship possible and for the many enlightning conversations we had.

Sam Owre was

giving me the chance to work on this project and for helping me throughout all my internship. I also had great help from Sam Owre and from Dr. Bruno Dutertre.

1.1 PVS Overview

PVS (Prototype Verification System) is an environment for specification and proving. The main purpose of PVS is to provide formal support for conceptualization and debugging in the early stages of the lifecycle of the design of hardware or software systems. In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable. The best way to analyze such an abstract specification is by attempting proofs of desirable consequences of the specification. Subtle errors revealed by trying to prove the properties are costly to detect and correct at later stages of the design lifecycle. he specification language of PVS is built on higher-order logic (functions can be treated like primitive types: functions can take functions as arguments and return them as values, quantification can be applied to function variables. Specifications can be constructed using definitions and axioms

1.2 The HACMS Project

1.3 Translating PVS

1.3.1 Parsing and typechecking PVS

These two task we leave to PVS native parser and typechecker.

The parser generates objects representing the expressions of the theory.

We only convert a subset of PVS. This subset is defined by a subset of expression objects we can translate. The objective is, of course, to be able to translate the maximum of (if not all) PVS expression objects.

1.3.2 Other translator

- Common Lisp (native) - Clean - Yices

2 PVS syntax

2.1 PVS Syntax

In Figure 11, we describe the subset of PVS we translate to C.

In Figure 12, we describe the object system used to represent them in Common Lisp. Some classes and some slots in the classes are voluntarily ommitted. For a full description of PVS parser representation, refer to [5].

2.2 Translator architecture

Describe here the Lisp functions and data structures

Skeleton

Expected input

Output objects
Assertions that we (try to) maintain
The translation from PVS to C is made following five main steps:

- Typechecking: The PVS typechecker perform a type analysis on the PVS code to associate a PVS type to each expression. This might generates some proof obligations (TCC).
- Lexical and syntactic analysis: The PVS parser transforms PVS code into a CLOS internal representation.
- Translation: The translator generates a different representation from PVS expressions and functions declarations. Typically, an expression e is translated into a tuple of four elements (t, n, i, d), where t represents a C type used to describe the expression, n is a string representing the expression, i is a list of instructions supposed to be executed prior to using n (initialisation of n) and d is a list of instructions to be executed when n isn't needed anymore (destruction of n).
- Analysis and optimizations: We run several analysis on the code representation. In particular, we determine the adapted C types, we try to avoid unnecessary copies and non destructive updates when possible using flow analysis and an enriched type system.
- Code generation: C code is generated (.c and .h files) and can be compiled using gcc and executed when linked with the garbage collector and the GMP library.

We first define a function T to translate an expression e.

$$T(e) = (\ T^t(e)\ ,\ T^n(e)\ ,\ T^i(e)\ ,\ T^d(e)\ )$$

T(`2`) = (`int`,"2", [], [])

T(`4294967296`) = (`mpz_t`, ? ,
[`mpz_init(?); |`
`mpz_set_str(?, "4294967296");`],
[`mpz_clear(?);`])

T(`lambda(x:below(10)):x`) = (`int*`, ? ,
[`? = malloc(10 * sizeof(int)); |`
`int i;|`
`for(i = 0; i < 10; i++)`
`?[i] = i;`]
[`free(?);`])

Figure 1: Translation examples: number expressions

It may occur that $T^n(e) =?$. In that case, the symbol ? appearing in $T^i(e)$ and $T^d(e)$ needs to be replaced by a proper variable name.

We then define two other operators:

- R wich take an expression and a type and may add an extra conversion in the instructions to make sure its result has the expected type. Also the result of this function has a proper name.
- S which take an expression, a type and a name. It makes sure that the given variable (type + name) is set to a value representing the expression.

2.3 A few translation rules

Translation rules :

```
number-expr "2"
(C-int, "2", [], [])

number-expr "12315468453213"
(C-mpz, nil,
        [mpz_t ~a; | mpz_t_init("12315468453213"); ],
        [mpz_clear ~a;])

application "f(e1, e2)"
(C-mpz, nil,
       [ instr(e1) | instr(e2)
                   | mpz(~a); | f(~a, e1, e2) ]
       [mpz_clear(~a);]
```

3 PVS type system

3.1 PVS Types

A PVS theory can be typechecked using the emacs interface `M-x typecheck` or calling the Lisp function `(tc name-theory)`. This first runs the PVS parser on the code and generates CLOS objects to represent it. Then, the PVS typechecker is run on this internal representation of the theory and tries to give a type to all expressions generating TCC when needed.

Here we describe how PVS types are represented in Lisp.

Figure 13 Figure 14

3.2 Translating types

PVS types:boolean, number, number_field, real, rational, integer, $A \to B$, restricted types below(10) := $\{x : \text{int} | 0 \leq x < 10\}$) enum datatype

This requires a type analysis to decide on the type of a PVS expression. For example the PVS `int` type can be represented by the `int`, `unsigned long` or `mpz_t` C types. In that case, we study the range of the expression to decide which types are allowed to represent it. Then we take the context in which the expression appears to decide. For instance in

```
incr(x:below(10)):int = x+1
```

the `x` expression, result of the function `incr` can always be represented by an `int` or `unsigned long` in C but we choose here to represent it using a `mpz_t`.

Intermediate type system : C-type with a flag : mutable (meaning that the expression it describes only has one pointer pointing to it.

```
int a = 2;      a : int[mutable]
int* a = malloc( 10 * sizeof(int*) );
```

destructive addition:

```
d_add(*mpz_t res, mpz_t[mutable] a, long b) {
  mpz_add(a, a, b);
  (*res) = a;
}
```

Rq : `d_add` is given a mutable `mpz_t`, meaning that it can modify it and is responsible for freeing it. It is also responsible for allocating memory for the result. Here it uses the memory to assign res.

Use an intermediate language :

```
( expr , C-type[mutable] )
```

Conversions and copies create mutables types (at a cost) : `a[mutable]_from_b`
[2]
C types:[3]

```
// integer and floating point types
[unsigned] char, int, long, double
type*  //arrays
char*  // strings
struct types  // structures with fields
enum types
short int, float, union, size_t // etc...
```

Listing 1: C types

Translation rules :

`subrange(a, b)`	`int` // if small enough `unsigned long` // if too big or needed for function call `mpz_t` // else
`int`	`mpz_t`
`rat`	`mpq_t`
`[below(a) -> Type]`	`(Ctype)*`

`T : TYPE = [#` x_i `:` t_i `#]`	`struct CT {`
	`...`
	`Ct_i x_i;`
	`...`
	`}; // These types must be declared`

`[Range -> Domain]`	C closure parameterized by the `Domain` return type.

Figure 2: Translation rules for PVS types

We can only translate a subset of all PVS types. What's missing ?

3.3 Translating PVS syntax

We can only translate a subset of PVS syntax. What's missing ?

3.4 Using a representation of the C language

Figure 16

4 Update expressions

It is a complicated problem to decide while compiling a functionnal language whether an update expression should be translated into a destructive or non destructive update in the target imperative language.

Update expressions are represented by PVS as `update- expr` objects.

$$E := \texttt{t with [e1 := e2]}$$

Problem : `t` is an expression typed as a function. Therefore it might be represented in C as an array (if domain type is `below(`n`)`. We want to know if we can update `t` in place to obtain a C object representing E or if we have to make a copy of `t`.

We consider a few solutions to this problem.

4.1 Pointer counting

Several systems rely on a reference couting garbage collectors. This family of garbage collectors has many advantages [4]. Along with its simplicity and the instantaneity of garbage identification, the one we are interested in is the possibility to determine when a local variable is the only pointer to a complex data structure. In that case, at the cost of a simple test, we can avoid copies and perform destructive updates.

The idea is to keep track of the number of pointers pointing to an array or a struct. If an array is referenced in several portions of the code (nested reference in other data structure, local variable in calling function, ...) then we mus be able, using the pointer counter, to perform all updates non destructively to avoid inconsistency.

We implement a very simple "Reference Counting Garbage Collector" as described in [4] and integrate it to the C code generated.

The GC consists on a hashtable of pointer counters that we maintain during the execution of the code. Each pointer to data allocated on the heap is a key in the hastable to which we associate an int counter as value. We then make sure that all memory allocations in the code make a call to the GC to "declare" the new memory.

`T* a = malloc( 10 * sizeof(int));`	`T* a = (T*) GC_malloc(10 * sizeof(int));` All memory allocation are handled by the GC to make sure every new reference on the heap is in the reference table and has a pointer counter associated to it.
`free(a);`	`GC_free(a);` This will decrement the reference counter on `a` and might free it if this counter is now 0.
`T* a = b;`	`T* a = (T*) GC( b );` The reference count on `b` is incremented to represent that the local variable `a` now also points to the structure `b` points to.

`t[0] = b;`	`GC_free( t[0] );`
	`t[0] = (T*) GC( b );`
	This time, we also make sure the reference counter of `t[0]` is decremented and `t[0]` has a chance to be freed if nothing else points to it.

This requires to build our own C garbage collector 3.

4.1.1 How to use it

The garbage collector mus be used for every manipulation of pointers to memory allocated on the heap. This occurs typically when representing PVS arrays or data structure. These rrays are created in the code.

When `A` points to an array (or `struct`) we want to update destructively, we first check if the pointer counter on `A` is 1. If so, we can update in place because only the local variable `A` points to the array.

```
struct entry_s {
   void*  pointer;
   int    counter;
   struct entry_s *tl;
};
typedef struct entry_s* entry;

struct hashtable_s {
   int    size;
   entry* table;
};
typedef struct hashtable_s* hashtable;

hashtable ht_create  ( int size );
int       ht_hashfunc( hashtable hashtable, void* pointer );
entry     ht_newentry( void* pointer );

hashtable GC_hashtable;
void      GC_start();
void      GC_quit();
entry     GC_get_entry( void* pointer );
void      GC_add_entry( entry e);
void      GC_new( void* pointer );
void*     GC( void* pointer );
int       GC_count( void* pointer );
void*     GC_malloc( int length, int size );
int       GC_free(void* pointer);
```

Figure 3: Garbage collector C header file: GC.h

However, we need to be carefull.

```
g(A:Array) :  int = f(A, A WITH [(0) := 3] )
```

should not be translated to

```
g(int* A) {
  A[0] = 3;
  return f(A, A);
}
```

for (at least) two reasons:

- The variable `A` is updated destructively but it is later used as a reference to the previous value of the array.
- `f` is given twice a pointer to the same data structure. Its reference counter should be incremented.

Instead we could flatten the expression 4.

But again, we are lucky here that `A` is the first argument of `f`. If the updated `A` were the first arguments, the update would have been done destructively. This is why the GC alone is not enoug. We need an analysis of the C code to determine whether a variable is goingto be used later in the code or not. cf 4.4 Analysis of the C code.

4.1.2 Pros and cons

The use of a garbage collector integrated in the C code seems like a good idea when translating a functionnal language to C. Using a pointer counting GC allows to

```
void main() {
   GC_start();

   int* A = GC_malloc(10, sizeof(int) );  // Pointer counter of A = 1
   int i;
   for(i = 0; i < 10; i++)  // Initialisation of A
      A[i] = i;             // Here A = lambda(x):x
   int* B = g( GC(A) );     // We need A further, we make sure that g knows
   int* C = A;              // main still has a pointer to A
   printf("Pointers to C = %d", GC_count(C) ); // equal to 2
   GC_free(B); // Frees B
   GC_free(C); // Only decrement the counter of C
   GC_free(A); // Frees A (and C)
   GC_quit();
}

g(int* A) {
  int* arg1 = GC(A);           // A and arg1 now both point to the array
  int* arg2;
  if (GC(A) == 1)              // This is false
     arg2 = GC( A );
  else {                       // The update must be done non destructively
     arg2 = GC_malloc( 10, sizeof(int) );
     int i;
     for(i =0; i < 10; i++)
         arg2[i] = A[i];
  }
  arg2[0] = 3;
  GC_free(A);                  // A is never used afterwards, we free it here
                               //(this requires an analysis of the C code)
  int* result = f(arg1, arg2);// A function is responsible for freeing its arguments
                               // (this is why we don't free arg1 and arg2)
  return result;
}
```

Figure 4: Example of the use of the GC

We need an analysis of the C code for two reasons:

- To `GC_free` variable as soon as they are not needed anymore. Otherwise copies that could be avoided are performed because an other (useless) pointer still points to the structure we're interested in.

  ```
  int* B = GC( A );
  update(B, 0, 1); // Can't be done destructively because A also points to
  GC_free(A);      // the same data as B
  f( GC(B) );      // f is given a variable with a reference counter of 2.
  GC_free( B );    // It might not be able to perform some update destructively
  ```

  Should be

  ```
  int* B = A;
  update(B, 0, 1);  // Can be done destructively
  f( B );           // f is given a variable with a reference counter of 1.
  ```

-

Every update require now tests and calls to hashtable functions. This is a small cost compared to the copying it may allow to avoid but no so small compared to a single in place update that could be decided by a code analysis.

Besides, the code gets much bigger since every update or copy requires the code to both destructive and non destructive operation and the if statement to decide which one to use.

Passing argument to function :

```
int* f(int* arg) {
   int* result;
   if ( GC_count(arg) == 1)
      result = GC( arg );
   else {
      result = GC_malloc(10, sizeof(int));
      int i;
      for(i = 0; i < 10; i++)
         result[i] = GC( arg[i] );
   }
   GC_free(arg);
   result[0] = 3;
   return result;
}
```

This add quite some code compared to the simple :

```
int* f(int* arg) {
   arg[0] = 3;
   return arg;
}
```

4.2 Using a different data structure

The Lisp code generated by PVS and used for example by the ground evaluator to compute PVS expressions represents PVS arrays with a more complex data structure than a simple array. It basically consists in an array and a replacement list. Every time an update on `(A, 1)` is performed, the result is a pointer to the same array `A` and a replacement list with an extra term: `(A, (0:=0) :: 1)`. When the length of the list becomes too big, we create a new array `A'` by applying the replacement terms to a copy of `A` and we return `(A', nil)`.

We could represent C data structure with a similar C structure. For example :

```
struct r_list {
   int key;
   int value;
   r_list* tl;
};
struct array_int {
   int *data;
   r_list* replacement_list;
};
```

Each structure represent the array `data` with the modifications contained in the linked list `r_list_int`.

As in the previous solution, we have the following issues:

- This adds some extra code
- This adds some extra computation. We need runtime tests for updates, and access to an element not requires reading the whole replacement list.
- This relies on lot on the GC.

4.3 Flow analysis on the PVS code

An other optimization would be to perform a analysis on the PVS variables to make sure an udate Pavol [1] suggests three analysis...

4.4 Analysis of the C code

This solution consist in performing an analysis on the C code internal representation before generating the actual output C code.

We use flags and two different version of the translated functions to translate update expressions (or dangerous function calls) into a destructive update as often as possible.

We define three flags:

- **mutable** means that the variable is the only pointer to the structure or array it points to. For instance if we have `f(A:Arr):Arr = A WITH [(0) := 0]` then when `f` is called in

  ```
  let A = lambda(x:int):x in let B = f(A) in B(0)
  ```

  we know that `f` can update `A` in place. We call the following version of `f`.

  ```
  int* f(int* A) {
    A[0] = 0;
    return A;
  }
  ```

- **safe** means that an occurrence of a variable is the last occurrence of that variable in the code. We need this flag to avoid updating destructively variables that appears later in the code. In the previous example, if we encounter

  ```
  let A = lambda(x:int):x in let B = f(A) in B(0) + A(0)
  ```

  we know we can't update `A` destructively and we call instead a non-destructive version of `f`:

  ```
  int* f(int* A) {
    int* res = malloc(...);
    for( i ...) res[i] = A[i];
    res[0] = 0;
    return res;
  }
  ```

- **duplicated** means that this expression may find itself nested in the result of the current function. For instance the identity function, `id(A:Arr):Arr = A`, has its argument flagged **duplicated** . Therefore when `id` is called we know that the result contains a pointer to its argument.

  ```
  ...
  int* A = malloc(...);
  [ init A somehow ]
  int* B = id(A);
  \\ From now on B and A point to the same array
  \\ For instance, A should probably not be modified in place
  ...
  ```

We want to ensure the following properties:

- Only function declarations and variables with type struct or array can be flagged **mutable** .
- Only a single occurrence of a variable may be flagged **safe** .
- Only expressions and arguments can be flagged **duplicated** .
- The last and only the last occurrence of a variable is flagged **safe** .
- Arguments of a non destructive function are never flagged **mutable** .
- A function is flagged **mutable** iif its return variable is flagged **mutable** .
- A variable may be flagged bang if it is created with a `copy`, `init_array`, `init_record` or is the result of a call to a function flagged **mutable** .
  It may not be flagged **mutable** if it is the result of a call to a function not flagged **mutable** .
- A call to a destructive function `f_d(` a_i`,` b_j`,` c_k `)` (where a_i are flagged **mutable** and b_j are flagged **duplicated** and c_k are not flagged) may only occurs if the following conditions on the arguments passed (A_i, B_j, C_k) are met:
  - All A_i are either calls to functions flagged **mutable** or variables flagged **mutable** and **safe** .
  - All B_j are either calls to functions or variables flagged **safe** or not flagged **mutable** .
  - If the function call is flagged **duplicated** , then all B_j are also flagged **duplicated** .
- If a variable is once flagged **duplicated** , then if it is an argument, this argument is also flagged **duplicated** .

Figure 5: Propeties of the flags

```
f(int* A, int* B) {          // A and B are both flagged duplicated
   if (A[0] == 0) {
      return B;
   } else {
      int* arg1 = copy(B);  // arg1 is flagged mutable and duplicated
      arg1[0] = arg1[0] - 1;
      f(arg1, A); // Both these occurrences of arg1 and A are flagged safe
   }
}

f_d(int* A, int* B) {  // A and B are both flagged mutable and duplicated
   if (A[0] == 0) {
      return B;
   } else {
      int* arg1 = B;     // No need to copy since B is mutable
                         // and never occurs afterwards
      arg1[0] = arg1[0] - 1;
      f_d(arg1, A);      // we can call f_d since the requirements are met:
   }                     //    both arg1 and A are flagged mutable
}
```

Figure 6: Example of the two different versions of a C function generated (stripped from GC instructions)

- Create the two versions of a function
- Flag all arguments **mutable** in destructive version
- Perform several passes and move flags to make sure the properties Figure 5 are verified.
- Modify the code if the flags allow it according to the rules defined in the Annex D.
- Redo the two previous steps until stabilization.

Figure 7: Algorithm

4.4.1 Algorithm

Each PVS function is translated into two different C functions:

- A "cautious" non destructive version whose arguments are never **mutable** and therefore never modifies the arguments in place, always making copies when necessary. This doesn't mean this function can't make destructive update. For instance locally created arrays (using `init_array`) will be flagged **mutable** and might be destructively updated, should the conditions be met.
- A destructive version which requires as many arguments as possible to be **mutable** and tries to do destructive updates as often as possible. This function only requires **mutable** arguments if it uses it destructively though.

In destructive versions of all functions : Flag all array arguments to "mutable". Then for each of these arguments : - If it never occurs destructively, then remove flag (function just read the arg) - If it occurs destructively, it can never occur at all AFTER. -¿ Need to define the order of evaluation of expression (easy rules on simple expressions) -¿ Need to be able to detect occurences of a name-expr -¿ Otherwise, unflag the arg

A variable V of type array is created in these cases:

- V `=` λ`x.e(x)` : V has bang type
- `update(`V`,` T`, key, value)` : V has bang type because this is basically a copy and a destructive update.
- `f(`V`,...)` `=` `...` : type of V depends on `f`.

In these case, it has always bang type.Or it can be set to an other referenced object.

- V `=` $T \rightarrow V$ (should have bang type iff T has !type too and never occurs afterwards). Happens in
- V `=` T`[i]` $\rightarrow$ depends on the target type of T.
- V `=` T`.field` $\rightarrow$ depends on the type of the field.

At first all updates are non destructive.

First pass : All array variables (actuals and local variables) found in the code are flagged. Local variables are flagged according to the previous rules and actuals are flagged **mutable** in destructive version and **not mutable** in non-destructive versions. In functions returning an array (or record type), the variable result is also flagged.

Other passes : Reading the code backwards, for every occurrence T of a variable flagged **mutable**:

- If it is found in a V `=` T instruction, then we give the bang type to V and remove bang type from T so that previous occurrences of T won't assume the uniqueness of the reference. This adds a new variable to the set of bang variables, hence the need to make several passes.
- If it is used in a V `=` `copy(`T`)` instruction, then we replace it with a V `=` T instruction and do as previous.
- If it is found in an `update(`V`,` T`, i, e`, then turn that into V `=` T`; destr_update(`V`, i, e)` .
- If it is a function call
- If we reach the declaration of a variable that is marked **mutable**, this means this variable is never read. In that case, we actually don't need it (unflag it I guess...).

At the end, when we have reached the transitive closure of this definition, if we reach the beginning of the function and some arguments are still bang, this means their bangness is never used, put the flag on that argument to `non mutable` and remove the instructions freeing that variable (reminder : mutable arguments of a functions are freed inside the function or are used in a mutable way and appear somewhere in the result (trapped in closures) or are freed in other function calls.

4.4.2 Algorithm

All variables have three flags: M (mutable), D (duplicated) and T (treated).

Init:All arguments of a destructive function are flagged (``M `= true, $D$ = false, $T$ = true`).All arguments of a non destructive function are flagged (`$M$ = false, $D$ = false, $T$ = true`).All other variables are flagged (`$M$ = false, $D$ = false, $T$ = false`).

Rules:When M is changed, T is set to `true`.When T is `true`, the flag M can only be set to `false`. This prevent infinite change of the flag M.The flag D can only be set to `true`.

Initialization:

We initialize a set M of mutable variables to all array arguments of a function f. We also initialize a set F of variables to free to M since f has the resposnability to free all variables flagged as mutable arguments.

We read the code backwards.T_i refer to variables that are in the set M.S_i refer to variables that are not in the set M.

S = T	$M \leftarrow M \cup \{S\} - \{T\}$ $F \leftarrow F \cup \{S\} - \{T\}$
S = update(S_2, key, value)	$M \leftarrow M \cup \{S\}$ $F \leftarrow F \cup \{S\}$
S = update(T, key, value)	$M \leftarrow M \cup \{S\} - \{T\}$ $F \leftarrow F \cup \{S\}$
S = g(T_i, S_i)	If the arguments of g don't allow g to be called destructively: $M \leftarrow M - \{T_i\}$ $M \leftarrow M \cup \{S\}$ if return type of g is **mutable** $F \leftarrow F \cup \{S\}$
S = g(T_i, S_i)	Otherwise: $\rightarrow S$ = g_d(T_i, S_i $M \leftarrow M \cup \{S\} - \{T\}$ $F \leftarrow F \cup \{S\}$
S = S_2[i]	$M \leftarrow M \cup \{S\} - \{T\}$ $F \leftarrow F \cup \{S\}$

All arguments of the function are flagged **mutable**

What is a destructive occurence :

$$E := \texttt{f(t with [e1 := e2] , t(0))}$$

order of eval : e1 and e2 (t can occur non destr) t (expression of an update : destr) t(0) (occurence of t (even non destr))

`f(x:Arr):int = g( h(t), t)` is destructively translated to

```
1 int f_d(int* t) {    // t has type ! since this is destructive f
2   int* arg1 = h(t); // h can't be called destructively because
3                     // even though t is !, it appears later (line 4)
4   int* arg2 = t;    // t is ! and never appears later => arg2 is !
5   return g( arg1, arg2); // arg2 is ! but g can only be called
6 }                        // destructively if arg1 is
```

Listing 2: Example

if g has type `[Array!  -> ?]` then t can't be destructive

if g has type `[Array -> ?]` then t can be destructive

First algorithm:

Need multiple passes as the flags disappear

4.5 Combination of solutions

We use the C code analysis to write some updates as destructive. However a few updates remain non destructive. For example:

If a function is called but requires its two argument to be **mutable** and only the first is **mutable** . Then the non-destructive version is called and the first argument gets copied even though it was **mutable** .

If we perform an update on T[i], our analysis doesn't tell if T[i] is **mutable** or **non-mutable** .

`update(A, key, value)`	`A[key] = value;`	`A` must be **mutable**
`set(A,` *expr*`)`	`A =` *expr*`;`	

`declare(A,` *expr*(i)`)`	`A = malloc(l * sizeof(T) );`
	`int i;`
	`for(i = 0; i < l; i++)`
	`  A[i] = i + 1;`	

`copy(A, B)`	`A = malloc(l * sizeof(T) );`
	`int i;`
	`for(i = 0; i < l; i++)`
	`  A[i] = B[i];`	

`init(A)`	`int* A;`	
`free(A)`	`free(A);`	
`base(str, (A, B, ...))`	`int aux = A[0] + B[1];`	`A` and `B` are only read.
`return`	`return result;`	

Figure 8: C instructions

`value(`*cste*`)`	`42`	
`variable(`*type*`,` *name*`)`	`name`	
`call(f,` *exprs*`)`	`f(` $expr_1$`, ...,` $expr_n$ `)`	

Figure 9: C expressions

To prevent that, we also perform a GC check. An update is actually a test wether an object is **mutable** or not and the appropriate update.

5 Conclusion

5.1 Difficulties and successes

5.1.1 Integrating the GMP library

In PVS, the `integer` represent the whole set $\mathbb{Z}$ of all relative numbers (and `rational` also describe $\mathbb{Q}$). To implement that in C, we need more than the finite types `int`, `long`, ...

We use the GMP library which introduces the types `mpz_t`and `mpq_t`. These types are pointers (actually arrays) to structures and they had to be used with caution (allocation, freeing, ...).

For example, a function returning a `mpz_t`should actually take a first `mpz_t`argument and set it to the return value. Its return type being `void`.

```
norm(x:int, y:int):int = x*x + y*y
```

```
void (mpz_t result, mpz_t x, mpz_t y) {
  mpz_t aux1;
  mpz_init(aux1);
  mpz_mul(aux1, x, x);
  mpz_t aux2;
  mpz_init(aux2);
  mpz_mul(aux2, y, y);
  mpz_add(result, aux1, aux2);
  mpz_clear aux1;
  mpz_clear aux2;
}
```

Figure 10: Example of the GMP library use

5.1.2 Working with new languages and tools

I had to learn three languages PVS, C, Common Lisp.

5.2 What's left to be done ?

Use a C structure to represent a closure

```
struct r_list_int {
   int (*body)(void* env, void* args);
   void* env;
   void* args;
};
```

5.3 My stay at SRI

Besides the conception and implementation of the PVS to C compiler, my stay at SRI International was rich in ??? events.

The first days of my stay were the occasion to discover PVS anc Coq as I started working on a translator Coq to PVS. With Robin, we also wrote as an exercise a basic linear algebra library.

I discovered Lisp the hard way while learning how the back end of PVS worked. I've wrote a Lisp parser to help me see clear in the huge code (classes definitions, inheritances and organization, function dependances, ...)

I also have had the chance to attend to the many interesting seminars SRI hosted every week. The SRI also organized a Summer School to which we were allowed to attend and which was very interesting.

Shankar never hesitated to include us in many project

I've been included in the HACMS project which was very interesting. With other: Correcting translator PVS to SMT-LIB

Draft

Discovering PVS : Translating Coq proofs to PVS PVS library for basic linear algebra

Robin project, HACMS Contest week-end 14-15 June Summer School Parsing Lisp code -¿ generate HTML architecture fileCorrecting translator PVS to SMT-LIB [1]

References

[1] Pavol Černý. Static analyses for guarded optimizations of high level languages.

[2] Jean-Christophe Filliâtre. Langages de programmation et compilation. MPRI, 2013-14. https://www.lri.fr/~filliatr/ens/compil/.

[3] Eric Huss. The c library reference guide. *Webmonkeys: A Special Interest*, 2004.

[4] Richard Jones and Rafael Lins. Garbage collection: Algorithms for automatic dynamic memory management, 1996. *John Wiliey & Sons Ltd., England*.

[5] N. Shankar and S. Owre. *PVS API Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 2003.

A PVS Syntax and CLOS representation

Expr ::= *Number* | *Name* | *Expr* (*Expr*$^{+}_{,}$)
| *Expr* *Binop* *Expr*
| *Unaryop* *Expr*
| *Expr* ‘ { *Id* | *Number* }
| (*Expr*$^{+}_{,}$)
| (# *Assignment*$^{+}_{;}$ #)
| *IfExpr*
| LET *LetBinding*$^{+}_{,}$ IN *Expr*
| *Expr* WHERE *LetBinding*$^{+}_{,}$
| *Expr* WITH [*Assignment*$^{+}_{;}$]

Number ::= *Digit*$^{+}$

Id ::= *Letter* *IdChar*$^{+}$

IdChar ::= *Letter* | *Digit*

Letter ::= A | ... | Z

Digit ::= 0 | ... | 9

IfExpr ::= IF *Expr* THEN *Expr*
{ELSIF *Expr* THEN *Expr*}* ELSE *Expr* ENDIF

Name ::= true | false | integer?
| floor | ceiling | rem | ndiv | even? | odd?
| cons | car | cdr | cons? | null | null?
| restrict | length | member | nth | append | reverse

Binop ::= = | \= | OR | \/ | AND | & | /\
| IMPLIES | => | WHEN | IFF | <=>
| + | - | * | / | < | <= | > | >=

Unaryop ::= NOT | -

Assignment ::= *AssignArg*$^{+}$ { := | |-> } *Expr*

AssignArg ::= (*Expr*$^{+}_{,}$) | ‘ *Id* | ‘ *Number*

LetBinding ::= { *LetBind* | (*LetBind*$^{+}_{,}$) } = *Expr*

LetBind ::= *Id* [: *TypeExpr*]

Figure 11: Syntax of the PVS subset of the translator

`expr` ⊂ `syntax` [*abstract class*]

type the type of the expression

`name` ⊂ `syntax` [*mixin class*]

id ..... the identifier

actuals .... a list of actual parameters

resolutions singleton

This is a mixin for names, i.e., `name-exprs`, `type-names`, etc.

`name-expr` ⊂ `name` `expr` [*class*]

`number-expr` ⊂ `expr` [*class*]

`tuple-expr` ⊂ `expr` [*class*]

exprs a list of expressions

`application` ⊂ `expr` [*class*]

operator an expr

argument an expr (maybe a tuple-expr)

`field-application` ⊂ `expr` [*class*]

id..... identifier

argument the argument

A field application is the internal representation for record extraction, e.g., `r‘a`

`lambda-expr` ⊂ `binding-expr` [*class*]

This is the subclass of `binding-expr` used for LAMBDA expressions.

`if-expr` ⊂ `application` [*class*]

`record-expr` ⊂ `expr` [*class*]

assignments a list of assignments

`update-expr` ⊂ `expr` [*class*]

expression. an expr

assignments a list of assignments

An update expression of the form `e WITH [x := 1, y := 2]`, maps to an `update-expr` instance, where the `expression` is `e`, and the `assignments` slot is set to the list of generated `assigment` instances.

`assignment` ⊂ `syntax` [*class*]

arguments. the list of arguments

expression the value expression

Assignments occur in both record-exprs and update-exprs.
The `arguments` form is a list of lists. For example, given the assignment `‘a(x, y)‘1 := 0`, the `arguments` are `((a) (x y) (1))` and the `expression` is `0`.

Figure 12: (Partial) CLOS representation of PVS syntax

B PVS type system and CLOS representation

TypeExpr	::=	*Name*
	|	*EnumerationType*
	|	*Subtype*
	|	*TypeApplication*
	|	*FunctionType*
	|	*TupleType*
	|	*CotupleType*
	|	*RecordType*
EnumerationType	::=	{ *IdOps* }
Subtype	::=	{ *SetBindings* | *Expr* }
	|	(*Expr*)
TypeApplication	::=	*Name Arguments*
FunctionType	::=	[FUNCTION | ARRAY]
		[–[*IdOp* :] *TypeExpr* $''^{+}_{,}$ -> *TypeExpr*]
TupleType	::=	[–[*IdOp* :] *TypeExpr* $''^{+}_{,}$]
CotupleType	::=	[–[*IdOp* :] *TypeExpr* $''^{+}_{+}$]
RecordType	::=	[# *FieldDecls*$^{+}_{,}$ #]
FieldDecls	::=	*Ids* : *TypeExpr*

Figure 13: Fragment of the PVS type system

type-expr ⊂ syntax	[*abstract class*]
type-name ⊂ type-expr name *adt*	[*class*]
subtype ⊂ type-expr *supertype* *predicate*	[*class*]
funtype ⊂ type-expr *domain* *range*.	[*class*]
tupletype ⊂ type-expr *types*	[*class*]
recordtype ⊂ type-expr *fields*	[*class*]

Figure 14: (Partial) CLOS representation of PVS types

C Intermediate language

```
Expr        ::= Number | Variable
              | Variable [ Variable ]
              | if ( Expr ) { Expr } else { Expr }
              | array( Variables )
              | Variable [ ( Variable ) := Variable ]
              | Variable [ ( Variable ) <- Variable ]
              | Function ( Variables )
              | set( Variable , Expr ); Expr
Variable    ::= Id
Function    ::= PrimOp | Id
PrimOp      ::= + | - | * | / | %
              | < | <= | > | >= | =
              | not | and | or | iff
FunctionDecl ::= Id ( Variable*, ) = Expr
Program     ::= FunctionDecl*, Expr
```

Figure 15: Syntax of the intermediate language

C.1 Operational semantic

A **value** is either an evaluated instruction ϵ, an integer or a reference $ref(i)$. The metavariable v ranges over the set V of all values.

An **evaluation context** E is an expression with an occurrence of a hole [] and is of one of the forms

1. `set(`x`, []); `e
2. []
3. $E1[E2]$, if E1 and E2 are evaluation contexts.

A redex is an expression of the following form

1. x
2. $X[y]$
3. `if (`x`) {`a`} else {`b`}`
4. `array(`x_1`, ..., `x_n`)`
5. X`[(`x`) := `y`]`
6. X`[(`x`) <- `y`]`
7. p`(`x_1`, ..., `x_n`)`
8. f`(`x_1`, ..., `x_n`)`
9. `set(`x`, `v`); `e

We define the **stack state**, s_1, as a function ranging over the set N of all variable names with values in V.

The **heap state** function, s_2 is mapping references $ref(i)$ to arrays of values, $V*$.

The **store** (or **state**) function, s, describing the state of the memory at a certain point in the execution is defined as the couple (s_1, s_2).

A program is list of function declarations followed by a closed expression. For each function with id f declared before the evaluation of the expression, we call f_i the arguments of this function (variables) and $[f]$ its body (expression).

The metavariable conventions are that x and y range over variables, X ranges over variables typed as arrays n ranges over numbers, p ranges over primitive function symbols, f ranges over defined function symbols, a, b and e range over expressions.

A **reduction** transforms a pair consisting of a redex and a store. The reductions corresponding to the redexes above are

- $< x, s > \longrightarrow < s_1(x), s >$
- $< x\texttt{[}y\texttt{]}, s > \longrightarrow < s_2(s_1(x))(s_1(y)), s >$
- $$\begin{cases} \dfrac{< a, s > \longrightarrow < v, s' >}{\texttt{if } x \texttt{ \{ } a \texttt{ \} else \{}b\texttt{\}}, s > \longrightarrow < v, (s_1, s_2') >} & \text{if } s_1(x) = 0 \\ \dfrac{< b, s > \longrightarrow < v, s' >}{\texttt{if } x \texttt{ \{ } a \texttt{ \} else \{}b\texttt{\}}, s > \longrightarrow < v, (s_1, s_2') >} & \text{otherwise} \end{cases}$$
- $< \texttt{array(}x_0\texttt{, ..., } x_{n-1}\texttt{)}, s > \longrightarrow < ref(m), (s_1, s_2 \uplus (ref(m) \mapsto (s_1(x_i))_{0 \leq i < n})) >$ where $s_2(ref(m))$ was not defined (meaning that $ref(m)$ is a "fresh pointer").
- $< X\texttt{[(}x\texttt{) := } y\texttt{]}, s > \longrightarrow < ref(m), (s_1, s_2') >$ where

$$\begin{array}{rcl} s_2(ref(m)) & \text{is not defined} & (ref(m) \text{ is a fresh pointer}) \\ s_2' & = & s_2 \uplus (ref(m) \mapsto s_2(s_1(X)) \uplus (s_1(x) \mapsto s_1(y))) \end{array}$$

- $< X\texttt{[(}x\texttt{) <- } y\texttt{]}, s > \longrightarrow < X, (s_1, s_2') >$ where

$$s_2' = s_2 \uplus (s_1(X) \mapsto s_2(s_1(X)) \uplus (s_1(x) \mapsto s_1(y)))$$

- $< p\texttt{(}x\texttt{, } y\texttt{)}, s > \longrightarrow < p(s_1(x), s_1(y)), s >$ for binary operators.
- $< p\texttt{(}x\texttt{)}, s > \longrightarrow < p(s_1(x)), s >$ for the `not` operator.
- $$\frac{< [f], (f_i \mapsto s_1(x_i), s_2) > \longrightarrow < v, (s_1', s_2') >}{< f\texttt{(}x_1\texttt{,...,}x_n\texttt{)}, (s_1, s_2) > \longrightarrow < v, (s_1, s_2') >}$$
- $< \texttt{set(}x\texttt{, } v\texttt{); } e, s > \longrightarrow < \epsilon, (s_1 \uplus (x \mapsto v), s_2) >$

An evaluation step operates on a pair $< e, s >$ consisting of a closed expression and a store, and is represented as $< e, s > \longrightarrow < e', s' >$. If e can be decomposed as a $E[a]$ for an evaluation context E and a redex a, then a step $< E[a], s > \longrightarrow < E[a'], s' >$. holds if $< a, s > \longrightarrow < a', s' >$ This is represented by the following rule.

$$\frac{< a, s > \longrightarrow < a', s' >}{< E[a], s > \longrightarrow < E[a'], s' >}$$

The reflexive-transitive closure of $\longrightarrow$ is represented $\longrightarrow^*$. The computation of a program is defined as the evaluation of its expression on an empty store: $< e, ([]_1, []_2) >$ (with $[]_i$ empty functions). If $< e, ([]_1, []_2) > \longrightarrow^* < e', s' >$ then we can prove that $e' \in V$ and the result of the computation is then defined as $eval_{s'_2}(e')$ where $eval_s$ is defined as follow:

$$\begin{aligned} eval_s : V &\longrightarrow E && (1)\\ n &\mapsto n \in \mathbb{N} && (2)\\ rem(k) &\mapsto (eval_s(u_i))_{0\leq i\leq n} \text{ with } (u_i)_{0\leq i<n} := s(rem(k)) && (3) \end{aligned}$$

Property : If $s_1(x) = ref(i)$ then s_2 is defined on $ref(i)$.

C.2 Live variables

We define the live variables of an expression :

$$\begin{aligned} LV\,(n) &:= \emptyset\\ LV\,(x) &:= \{x\}\\ LV\,(X\texttt{[}x\texttt{]}) &:= \{x\}\\ LV\,(\texttt{if (}x\texttt{) \{ }a\texttt{ \} else \{ }b\texttt{ \}}) &:= \{x\} \cup LV(a) \cup LV(b)\\ LV\,(\texttt{array(}x\texttt{, }n\texttt{, }e\texttt{)}) &:= LV(e) - \{x\}\\ LV\,(X\texttt{[(}x\texttt{) := }y\texttt{]}) &:= \{X, x, y\}\\ LV\,(X\texttt{[(}x\texttt{) <- }y\texttt{]}) &:= \{X, x, y\}\\ LV\,(f(x_1\texttt{, ...,}\ x_n)) &:= \{x_i | 1 \leq i \leq n\}\\ LV\,(\texttt{set(}x\texttt{, }a\texttt{); }e) &:= LV(a) \cup (LV(e) - \{x\}) \end{aligned}$$

The translator from PVS to that intermediate language guarantees the following properties

- All arguments passed to a function are **different** variables.
- The first argument of `init_array` or `update` is always the first occurence of a variable typed as array. Variables typed as array first appear as the first argument of a `init_array` or `update` instruction.
- Before the analysis, no `destr_update` are used.

For the need of the analysis, we enrich the state function with a reference counter $c : V \to \mathbb{N}$.

```
Expr          ::= Number | String
                | Function ( Exprs )
                | Pointer

Pointer       ::= Variable
                | Variable . Id
                | Variable [ Expr ]

Variable      ::= ( Type , Id )

Type          ::= int | unsigned long int
                | mpz_t | mpq_t
                | array( Type , Number )
                | struct( Id )

Instruction   ::= decl( Variable )
                | free( Variable )
                | if ( Expr ) {
                    Instructions
                  }else {
                    Instructions
                  }
                | MPZFunction ( Variable [ , Exprs] )
                | init_array( Variable , Instructions , Expr )
                | init_record( Variable , Instructions , Exprs )
                | set( Pointer , Expr )

Function      ::= + | * | ...
                | Id

MPZFunction   ::= mpz_set_str | mpz_add | ...

FunctionDecl  ::= Id ( Variables ) : Type =
                  Instructions
                  [ return Expr] ;

StructDecl    ::= struct Id : Types

Types         ::= [ Type [ , Types] ]

Exprs         ::= [ Expr [ , Exprs] ]

Variables     ::= [ Variable [ , Variables] ]

Instructions  ::= [Instruction ; [Instructions] ]
```

Figure 16: Syntax of the representation language (representation of a subset of the C language)

D Rules

	`A` **safe**	`A` not **safe**

`A` **mutable**	Replace every occurence of the variable `B` by the variable `A`	`B = GC_malloc(...);`
		`for(i ...) {`
		`  B[i] = A[i];`
		`}`

`A` **non-mutable**	`if (GC_count(A) == 1) {`
	`  B = A;`
	`} else {`
	`  B = GC_malloc(...);`
	`  for(i ...) {`
	`  B[i] = A[i];`
	`  }`
	`}`
		`B = GC_malloc(...);`
		`for(i ...) {`
		`  B[i] = A[i];`
		`}`

Figure 17: Rules for `copy(B, A)`

	`A` **safe**	`A` not **safe**

`A` **mutable**	Replace every occurence of the variable `B` by the variable `A`	`B = GC_malloc(...);`
		`for(i ...) {`
		`  B[i] = A[i]`
		`}`

`A` **non-mutable**	Replace every occurence of the variable `B` by the variable `A`	`B = GC( A );` If `B` is flagged **duplicated** then `A` must be too.

Figure 18: Rules for `set(B, A)`

	`A` **safe**	`A` not **safe**
`A` **mutable**	`f_d(A)`	`f(A)`
`A` not **mutable**	`f(A)`	`f(A)`

Figure 19: Rules for `f(A)` with `A` flagged **mutable** in the destructive version

	`A` **safe**	`A` not **safe**
`A` **mutable**	`f(A)`	`copy(B, A)` `f(B)`
`A` not **mutable**	`f(A)`	`f(A)`

Figure 20: Rules for `f(A)` with `A` flagged **duplicated**

E Examples

PVS code	Intermediate language code	C code generated

`f(A:Arr):Arr = A`	`f: ( (A, int*) ) -> int*`
	`set(result, A)`
	`return(result)`
		`int* f(int* A) {`
		`  return A;`
		`}`

`f(A:Arr):Arr =`
`  let B = A in B`
	`f: ( (A, int*) ) -> int*`
	`set(B, A)`
	`set(result, B)`
	`return(result)`
		`int* f(int* A) {`
		`  return A;`
		`}`

`f(A:Arr):Cint =`
` let B = A in`
`  A(0) + B(0)`
	`f: ( (A, int*) ) -> int`
	`set(B, A)`
	`set(result, +( A(0), B(0)`
	`) )`
	`return(result)`
		`int* f(int* A) {`
		`  int* B = (int*) GC( A );`
		`  int result = A[0] + B[0];`
		`  GC_free(B);`
		`  GC_free(A);`
		`  return result;`
		`}`

`f(A:Arr):Arr =`
`  let B = A in`
`  A WITH [(0) := B(0) ]`
	`f: ( (A, int*) ) -> int`
	`set(B, A)`
	`set(L, 0)`
	`set(R, B(0) )`
	`update(result, A, L, R)`
	`return(result)`
		`int* f_d(int* A) {`
		`  int* B = GC_malloc(...);`
		`  for(i ...)`
		`  B[i] = A[i];`
		`  int L = 0;`
		`  int R = B[0];`
		`  int* result = GC( A );`
		`  result[L] = R;`
		`  GC_free(B);`
		`  GC_free(A);`
		`  return result;`
		`}`

Figure 21: Examples of setting variables

<table>
<tr><th>PVS code</th><th>Intermediate language code</th><th>C code generated</th></tr>
<tr>
<td>

```
f(A:Arr):Arr =
  A WITH [(0) := 0]
```

</td>
<td>

```
f:  ( (A, int*) ) -> int*
set(L, 0)
set(R, 0)
copy(result, A)
update(result, L, R)
return(result)
```

</td>
<td>

```
int* f(int* A) {
  int L = 0, R = 0;
  int* result;
  if( GC_count(A) == 1 ) {
    result = GC( A );
  } else {
    result = GC_alloc(...);
    for(i ...)
      result[i] = A[i];
  }
  result[L] = R;
  GC_free( A );
  return result;
}

int* f_d(int* A) {
  int L = 0, R = 0;
  A[L] = R;
  return A;
}
```

</td>
</tr>
<tr>
<td>

```
f(A:Arr):Arr =
 let B = A WITH[(0):=0]
 in A WITH[(0) := B(0)]
```

</td>
<td>

```
f:  ( (A, int*) ) -> int*
set(L1, 0)
set(R1, 0)
copy(B, A)
update(B, L1, R1)
set(L2, 0)
set(R2, B(0))
copy(result, A)
update(result, L1, R1)
return(result)
```

</td>
<td>

```
int* f(int* A) {
  int R1 = 0, L1 = 0;
  B = GC_alloc(...);
  for(i ...)
    B[i] = A[i];
  B[L1] = R1;
  int R2 = 0, L2 = B[0];
  result = GC_alloc(...);
  for(i ...)
    result[i] = A[i];
  result[L2] = R2;
  GC_free(A);
  GC_free(B);
  return result;
}

int* f_d(int* A) {
  int R1 = 0, L1 = 0;
  B = GC_alloc(...);
  for(i ...)
    B[i] = A[i];
  B[L1] = R1;
  int R2 = 0, L2 = B[0];
  A[L2] = R2;
  GC_free(B);
  return A;
}
```

</td>
</tr>
</table>

Figure 22: Examples of copying variables