ÉCOLE POLYTECHNIQUE
PROMOTION X2011
Férey Gaspard

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

Translating PVS to Efficient C

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

1er septembre 2014

Abstract

PVS (standing for Prototype Verification System), is an Open Source project developed by CSL at SRI International and aiming to be both a semi-automated theorem prover providing formal support for conceptualization and debugging in the early stages of the design of hardware or software systems and a programming language. The evaluation of PVS expressions relies so far on a build-in PVS interpreter based on Common Lisp, called "Ground Evaluator". In order to allow the integration of PVS code as well as its fast execution for debugging and testing purposes, we describe here a translator of a subset of PVS to the language C.

The update of aggregate data structure, such as arrays, are frequent in functional programs and requires copying before being updated which is a significant source of space/time inefficiencies. However the execution of updates by copying is often redundant and could be safely implemented by means of destructive, in-place updates in an imperative program. We describe a simple method for analyzing and replacing the safe updates in an imperative program with destructive, in-place update. This method has been implemented to optimize the PVS translator.

Acknowledgements

I would like to thank my supervisor, Natarajan Shankar, for his help, explanations and suggestions as well as for the many enlightening discussions we had during this internship. I also thank Sam Owre for his explanations of the PVS API and Common Lisp in general, Robin Larrieu, from Polytechnique who shared an office and a lot of good ideas with me. I thank all of my teachers from LIX who made this internship possible, with a special mention to Stéphane Graham-Lengrand and Benjamin Doerr who recommended me. Finally, I thank all the people at the CSL, for their welcome, the interesting discussions I had with them, and for creating an exciting and inspiring environment for work. A special thank Lori Truitt for all the help she provided with administrative paperwork.

Contents

1 Introduction

In this section, we quickly describe PVS and the previous effort made in its translation. Then, we give in Section 3 an overview of the different steps in the process of translating a functional, higher order, specification language into an imperative, low-level language such as C. We provides a few detail on the implementation of our translator and give examples of its execution. In Section 3 we describe the issue of the translation of updates expressions and describe a few solutions, two of them being implemented in the translator. Finally we focus on the implemented static analysis in Section 4 and try to prove its correctness.

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 life cycle 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 life cycle.

The 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 (universal and existential) can be applied to function variables. Specifications can be constructed using definitions and axioms.

PVS typechecking is not decidable and thus whenever the typechecker encounters a non trivial constraint for an expression to be type-correct, it relies on *Type-Correctness Condition* (TCC), a formula that has to be proved by the user for the expression to actually be typed correct. For instance, trying to divide by some variable x will automatically generate a TCC for the user to prove that that x, in the context of the call, can not be zero.

A PVS file consists in one or several theories, that are somewhat similar to modules in OCaml and Python, or to packages in Java. The body of a theory consists in a serie of declarations of the form `name:sort = def` where `sort` is the “sort” of the declaration and `def` is a PVS expression which can be a term expression (syntax described in Figure 9) or a type expression (syntax described in Figure 11). When the `= def` part is omitted, the declaration is uninterpreted and the name now refers to some abstract value of `sort`. See Figure 20 for an example of a theory.

The sort of the declaration can be:

- A PVS type expression (lines 15-17 in the PVS example), in which case the declaration defines a new value of that type. A TCC is generated to prove that the definition is of the given type unless no definition is given. In that case, a TCC might be generated to prove the type is non empty.
- `TYPE` or `NONEMPTY_TYPE` (or `TYPE+`), in which case the declaration defines a new type. A TCC might be generated to guarantee the non emptiness.
- A theorem-like keyword such as `THEOREM` or `LEMMA`, in which case the declaration defines a new theorem. A proof obligation is generated, and the theorem can be used in proofs in the remaining declarations in the file.
- An axiom-like keyword such as `AXIOM` or `POSTULATE`, in which case the declaration introduces a new axiom. No proof obligation is generated and the proposition can be used in

proofs in the remaining declarations in the file. This can obviously introduce inconsistencies, and should be used with much care. It is often used in conjunction with uninterpreted declarations to define types or terms axiomatically instead of providing a definition.

In addition to that, the PVS `DATATYPE` mechanism allows to define recursive sum types with constructors, accessors, and recognizers. There is no polymorphism in PVS, however *theory parameters* allow to have some sort of genericity. Finally, another mechanism that we left out and that is worth mentioning is the `IMPORTING` keyword which has a visibility effect and is required in the PVS implementation to be able to talk about a theory inside another theory.

1.2 Why translate PVS ?

Specification languages such as PVS are made to be expressive rather than executable allowing the conceptualization, design, modelization and analysis of complex systems described at a higher level than a programming language.

However such systems are themselves made to be later executed and integrated into physical systems. For that reason, a model of a system needs to be turned into executable code implemented into a programming language. This can be done manually with the risk of making human mistakes in the translation process or this can be left to a automated translator which would preserve the high insurance properties that were proven on the model. Besides, the ability to run a model during its development phase allows debugging and testing.

The HACMS Project

The DARPA-funded project High Assurance Cyber-Military System (HACMS) aims to produce systems and software with proved reliability and security. It uses the abstraction of nodes and topics to describe the different components and communication channels. A PVS model of these nodes and topics allows to prove security properties.

This is a good example of system which requires an implementation and integration of the proven algorithms. For that purpose, a translator PVS to C would be very helpful since most of the ROS systems use the C language.

Other translators

Translating PVS was already done for different purposes.

- PVS come with a native PVS to Common Lisp translator. This is used to run PVS programs for testing and debugging purposes. Since PVS's API relies heavily on the Common Lisp language, it is very easy to run PVS code within Emacs. The "Ground Evaluator" uses the generated Common Lisp code to evaluate PVS expressions. It provides a more user-friendly interface for the PVS translator by being integrated into Emacs and translating simple Common Lisp expression back to PVS.

- PVS expressions can also be translated to Yices's specification language syntax. Yices is an efficient SMT solver developed at SRI by Bruno Dutertre. This translation is a way to "plug" the Yices solver into PVS, allowing automated proof of theorems and TCCs.

- PVS was translated to the Clean language.

- Lately PVS was also translated to the SMT-LIB language (work in progress) for standardization purposes. This would allow every SMT solver able to process SMT-LIB standard inputs such as Yices 2, to be plugged into PVS for further proof automation.

2 Translating PVS

A translator is a program taking the source code of program P written in the programming language $\mathcal{L}_A$ as an entry and generating the code of an other program Q in the target language $\mathcal{L}_B$ as an output. In our case, $\mathcal{L}_A$ is a subset of PVS and $\mathcal{L}_B$ is the language C. It is expected from a translator to:

- Never fail if the entry is the code of a valid program in the input language, $\mathcal{L}_A$. However the translator might declare being only able to translate a fragment of the language and restrict its input language. In our case, we expect input programs to be written using the syntax in Figure 9, to be parsed and typechecked using PVS without error and finally, we expect that TCCs generated by PVS can be proven.

- Generate a valid program in the target language $\mathcal{L}_B$. In our case, some of the key requirements are that the generated code can be compiled using the GNU Compiler Collection (gcc.gnu.org/) and never raises a segmentation fault error during its execution.

- For all entry x, $P(x)$ and $Q(x)$ return the same result (correctness). More precisely, since both programs take different data structure as entries, if x_A and x_B have the same mathematical evaluation respectively in the languages $\mathcal{L}_A$ and $\mathcal{L}_B$ then $P(x)$ and $Q(x)$ also have the same mathematical evaluation.

2.1 Translator's architecture

The translation from PVS [Owr+98] to C follows these main steps:

- Typechecking: The PVS typechecker [Owr+99b] perform a type analysis on the PVS code to associate a PVS type to each expression. This might generates some proof obligations (TCC). The user of the translator has to make sure that the PVS code can be correctly typechecked and that all TCC can be proven.

- Lexical and syntactic analysis: The PVS parser transforms PVS [Owr+99a] code into a CLOS internal representation.

  In Figure 9, we describe the syntax of the subset of PVS we are currently able to translate to C. In Figure 10, we describe the Common Lisp Object System architecture used by PVS to represent them in Common Lisp. Some classes and some slots in the classes are voluntarily omitted. For a full description of PVS parser representation, refer to [SO03].

- Translation: The translator flattens all PVS definitions to generates a program in an intermediate language which heavily relies on the use of intermediate variables to store the values of every expression. Besides, this form allows a simpler static analysis. The translation is briefly described in subsection 2.3. The syntax of this language is described Figure 13.

- Static analysis: The intermediate language is analyzed and stripped from some of its unnecessary copies and non destructive updates using flow analysis. This analysis inspired from Shankar [Sha02] and Cerny and Shankar's [ČS06] previous analysis of PVS is described with more detail in Section 4.

- Optimizations: Several simple analysis are performed to determine, for instance, where to declare and free variables as well as the most adapted C types to use. The output is a more complete and closer to C version of the intermediate language. The type translation is described in the subsection 2.2. The code generated from that step can be described by the syntax in Figure 14.

- 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. The C syntax is described in [Hus04] and the GMP library reference can be found at https://gmplib.org/manual/.

2.2 Translating PVS type

2.2.1 PVS type system

A PVS theory needs to be typechecked using the emacs interface `M-x typecheck` or calling the Lisp function `(tc name-theory)`. This 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.

The (simplified) syntax for PVS types is described in Figure 11. PVS allow a base types such as `booleans` or `numbers`. Then more complex types can be defined such as sets, tuples, datatypes or functions with range and values over other types. Types can also be restricted into subtypes using predicates. For example, `integers` are defined as a subtype of `rationals` with the predicate `integer_pred`. This way, when an expression is passed to a function ranging over integers, the typechecker generates the TCC "argument must verify the predicate `integer_pred`". That TCC must then be proven (it is often proven automatically for such simple TCCs).

The Figure 12 describes how PVS type system is represented in CLOS.

2.2.2 C types

The C language [Hus04] has a few base types to represent bounded integers (`int`, `long`, etc). It allows to define enumerated types, structures containing several fields with different types. The variables with a pointer type have a memory address as a value. They can be used to reference arrays dynamically allocated in the heap.

To represent big integers or rationals, we use the GMP library which introduces a few other types.

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

Listing 1: C types

2.2.3 Translation rules

The translation of PVS types requires a type analysis to decide on the type of a PVS expression.

For instance the translation of the PVS `int` type can be done using the `int`, `unsigned long` or `mpz_t` C types. In that case, we need to study the range of the expression to decide which types are best to represent it. Then we take the context in which the expression appears to decide. For instance when a variable `x` is typed with a subtype that bounds the range of the values `x` can take, we can safely represent it with a C bounded type.

```
incr(x:below(10)):int = x+1
int incr(int x) { return x+1; }
```

In that case, we not only decide to represent `x` with an integer but also the expressions `1` and `x+1` as well as the return value of `incr`. This requires an range analysis to realize that `1` $\in [1;1]$ and `x+1` $\in [1;10]$ can both be represented by the `int` type.

When such optimizations are impossible we have to rely on bigger base types or use the GMP types which can generate a much less readable code (see Section 5, Figure 8) and often requires a few conversions.

We decide represent functions ranging over integer bounded type with index starting at 0 with arrays and other with closures. It is the responsibility of the user who needs an efficient representation with C arrays to use such types.

We describe here a few 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 1: Translation rules for PVS types

2.3 Translating PVS syntax

Whereas PVS syntax relies on expressions built from other expressions in a tree-like structure, the C language relies on a instructions, memory allocation and the use of variables. Even though some simple PVS expression (for instance `2*(x+y)` for `x` and `y` typed as small integers) can be directly translated to C expressions (`2*(x+y)` with `x` and `y` typed as `int`), most of the time to represent a PVS expression in the C language, we need to build it and store it into a variable. For instance `X[0] + 1` should be translated to

```
mpz_t aux = X[0];
mpz_t one;
mpz_init_set_ui(one, (unsigned long int) 1);
mpz_t res;
mpz_init( res );
mpz_add(res, aux, one);
mpz_clear(aux);
mpz_clear(one);
[...] // Use res here, its value represent X[0] + 1
mpz_clear(res);
```

So to obtain a C expression (variable) representing a PVS expression, we usually need a few instructions to initialize it, the variable (or C expression) itself and a few instructions to destroy the variables created.

Typically, an expression e is going to be translated into a tuple of four elements (T, E, I, D), where T represents a C type used to describe the expression, E is a simple expression (a variable, a function call, an access to an array, etc), I is a list of instructions to be executed prior to using E, the initialization of the expression. Finally D is a list of instructions to be executed when E isn't needed anymore, the destruction of E.

Main functions

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

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

where $T^n(e)$ might be a hole: ?. In that case, the instructions $T^i(e)$ and $T^d(e)$ might contain occurrences of the hole. From that function, we define a similar function

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

which decides the type of the C expression returned. This function uses the result of T and may perform conversions to make sure its result has the expected type. Also the result of this function is never a hole. Finally the function

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

imposes the C expression to be a variable with the given name and type.

Example

As an example, we describe here what would be the trace of the execution of the translator on the following PVS expression.

$$E := \texttt{lambda(x:below(10)):x WITH (1) := 5}$$

We suppose that the static analysis decides to make the update destructively. The output would be, before the optimization:

```
mpz_t* A;
A = malloc( 10 * sizeof(mpz_t) );
for (i=0; i<10; i++) {  mpz_init(A[i]);  }
for (x=0; x<10; x++) {
   mpz_init(A[i]);
   mpz_set_ui( A[x], x );
}
int N;
N = 1;
mpz_t R;
mpz_init(R);
mpz_set_ui(R, 5);
mpz_set(A[n], R);
mpz_clear(R);
 [...] // use A here
for(x=0; x<10; x++)  mpz_clear(A[i]);
free(A);
```

Figure 2 is the description of the trace of the translator.

2.4 Optimization of the intermediate languages

In order to perform a few analysis, do not translate PVS directly to C but use two intermediate languages instead.

1. The first of these languages, which syntax is described Figure 13, is meant to be simple, close enough to C so that optimization performed on it directly translate to C optimization and expressive enough to represent the whole subset of PVS we are interested in.

   On that language, we perform the analysis consisting in replacing non destructive updates with destructive updates as often as possible. The algorithm is described in Section 3 and the analysis itself is described in Section 4.

To compute	Needed	I	[T] E	D
$R(E, \texttt{mpz\_t*})$	$S_1 := S(\lambda\texttt{x.x}, \texttt{mpz\_t*}, \texttt{A})$ $S_2 := S(\texttt{1}, \texttt{int}, \texttt{N})$ $S_3 := S(\texttt{5}, \texttt{mpz\_t}, \texttt{R})$	`mpz_t*A;` S_1^i `int N;` S_2^i `mpz_t R;` S_3^i `mpz_set(A[N],R);` S_1^d S_2^d S_3^d	`A`	S_1^d
$S(\texttt{1}, \texttt{int}, \texttt{N})$	$R_1 := R(\texttt{1}, \texttt{int})$	R_1^i `N =` R_1^e`;` R_1^d	`N`	
$R(\texttt{1}, \texttt{int})$	$T_1 := T(\texttt{1})$	T_1^i	T_1^e	T_1^d
$T(\texttt{1})$			`int 1`	

$S(\texttt{5}, \texttt{mpz\_t}, \texttt{R})$	$T_1 := T(\texttt{5})$	T_1^i
		`mpz_init(R);`
		`mpz_set_ui(R,` T_1^e`);`
		T_1^d
			`R`	`mpz_clear(R);`

$T(\texttt{5})$			`int 5`	

$S(\lambda\texttt{x.x}, \texttt{mpz\_t*}, \texttt{A})$	$S_x := S(\texttt{x}, \texttt{mpz\_t}, \texttt{A[x]})$	`A = malloc(10...);`
		`for(x=0...10)`
		S_x^i
			`R`	`for(x=0...10)`
				S_x^d
				`free(A);`

$S(\texttt{x}, \texttt{mpz\_t}, \texttt{A[x]})$	$T_1 := T(x)$	`mpz_init(A[x]);`
		T_1^i
		`mpz_set_ui(A[x],`
		T_1^e`);`
			`A[x]`	T_1^d
				`mpz_clear(`
				`A[x]);`

$T(x)$			`int x`	

Figure 2: Example of the translator's execution trace

2. The second language is very close to C but allows to manipulate the code easily and perform a few other optimizations.
    - Determining the adapted C type (especially for integers and function types) to represent variables and adding call to conversion function when necessary.
    - Detecting PVS primitive function calls and the adapted C function to call.
    - Declaring the structures needed.
    - Optimizing the position of declaration and destruction of pointer variables.

3 Update expressions

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

PVS update expressions are represented in CLOS as `update-expr` objects

$$E := \ T \ \texttt{with [(}e1\texttt{) := } e2 \texttt{]}$$

where T is an expression typed as a function and therefore 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 and update the copy.

We discuss here a few solutions to this problem and describe how they are or could have been implemented in the translator.

3.1 Pointer counting

Several systems rely on a reference counting garbage collectors. This family of garbage collectors has many advantages [JL]. 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 safely avoid copies and perform destructive updates since we are sure no other variables elsewhere in the code is going to be affected by this in place update. We however need to make sure the variable we are updating itself is never used later.

The idea is to keep track of the number of pointers pointing to an array or a struct. We can detect, by checking the pointer counter, if an array is referenced in several portions of the code (nested reference in other data structure, local variable in calling function, ...) and then perform all updates non destructively to avoid inconsistency.

We implement a very simple "Reference Counting Garbage Collector" as described in [JL] and integrate it to the C code generated according to the rules described Figure 3.

The GC consists in 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 hashtable 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. The GC we implemented is described Figure 4.

3.1.1 How to use it

The garbage collector must be used for every manipulation of pointers to memory which was dynamically allocated on the heap. This occurs typically when representing PVS arrays or data structure.

When `A` points to an array (or `struct`) we want to update destructively, we first have to 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.

However, we need to be careful.

```
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);
}
```

<table>
<tr><td>

```
T* a =
  malloc(10 *
    sizeof(int));
```

</td><td>

```
T* a = (T*) GC_malloc(10 * sizeof(int));
```

All memory allocation in the heap are handled by the GC's `GC_malloc` to make sure every new reference on the heap is in the reference table and has a pointer counter associated to it.
</td></tr>
<tr><td>

```
T* a = b;
```

</td><td>

```
T* a = (T*) GC( b );
```

The reference count on `b` is incremented with `GC` to represent that the local variable `a` now also points to the structure `b` points to.
</td></tr>
<tr><td>

```
T* f(T* a) {
  T* b = a;
  T* c = b;
  return c;
}
```

</td><td>

```
T* f(T* a) {
  T* b = (T*) GC( a );
  T* c = (T*) GC( b );
  GC_free(b);
  GC_free(a);
  return c;
}
```

The `GC_free` instruction will decrement the reference counter of its argument and might free it if this counter is now 0. We use it to decrement the pointer of all local pointers (including arguments) before they are deallocated from the stack. However we do not decrement the return variable since we don't want it to be freed (dangling pointer risk).
</td></tr>
<tr><td>

```
t[0] = b;
```

</td><td>

```
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.
</td></tr>
<tr><td>

```
{
  T** t = a;
  [...]
}
```

</td><td>

```
{  T** t = GC( a );
   [...]
   GC_free( t );
   if (GC_count(t) == 1)
     for (i = 0; i < ...; i++)
       GC_free( t[i] );
   GC_free( t );   }
```

When an array of arrays is freed, we need to make sure we decrement everything it was pointing to.
</td></tr>
</table>

Figure 3: GC instructions

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.

Below are a few rules a good use of the GC should follow (see Figure 5).

```c
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 4: Garbage collector C header file: GC.h

1. All dynamically allocated memory on the stack should be done using the GC.
2. All function is responsible for freeing all its arguments. Indeed, the local variable implicitly created to represent the argument is itself a pointer to the structure.
3. All argument passed to a function must have its counter incremented via the GC.

However a few optimizations are possible. For instance if a variable has its counter incremented before being passed to a function and is then freed right after, then it can be directly passed and rely on the function call to free it.

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 enough. We need an analysis of the C code to determine whether a variable is going to be used later in the code or not (**safe** occurrence, cf 3.4 Analysis of the intermediate language).

3.1.2 Pros and cons

The use of a garbage collector integrated in the C code seems like a good idea when translating a functional language to C. Using a pointer counting GC allows to dynamically allocate memory on the heap and pass or return such dynamically allocated object without worrying about where and when they are going to be freed.

We however need an analysis of the C code for several 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.

```
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 5: Example of the use of the GC

```
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 now 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. For instance, passing arguments to a function adds quite some code

GC use	Static analysis optimization

GC use:

```
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;
}
```

Static analysis optimization:

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

3.2 Using a more adapted 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, l)` is performed, the result is a pointer to the same array `A` and a replacement list with an extra term.

$$\texttt{T} \Longrightarrow \texttt{(A, l)}$$
$$\texttt{T[(0) := 0]} \Longrightarrow \texttt{(A, (0:=0) :: l)}$$

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

```
struct array_int {
   int *data;
   r_int_list* replacement_list;
};
```

```
struct r_int_list {
   int key;
   int value;
   r_list* tl;
};
```

When the replacement list becomes too long (longer than $\tau(n)$), we create a new array `A'` by applying the replacement terms to a copy of `A` and we return `(A', nil)`. The hope is that by the time we need to perform this copy, nothing else points to the old array so that it is garbage collected immediately. The trade off can be summarized with the worst case complexities:

	Copied array	New data structure
Update time	$O(n)$	$O(n)$ (if copying is needed) but most of the time $O(1)$.
Update space	$O(n)$	$O(n)$ (if copying is needed and the old array is not garbage collected) but most of the time $O(1)$.
Access time	$O(1)$	$O(\tau(n))$ since we need to read the replacement list.

Best case scenario, all arrays are always immediately garbage collected after an update and we get these mean complexities

	Copied array	New data structure	Destructive update
Update time	$O(n)$	$O(n/\tau(n))$	$O(1)$
Update space	$O(n)$	$O(1)$	0
Access time	$O(1)$	$O(\tau(n))$	$O(1)$

We have the following issues:

- This adds some extra code both to implement the new data structures and algorithms and to use them.
- This adds some extra run time for reading accesses which require reading the whole replacement list.
- This relies a lot on the GC.

3.3 Flow analysis on the PVS code

An other optimization would be to perform an analysis on the PVS expressions. Shankar [Sha02] and later Cerny and Shankar's [ČS06] suggest several analysis based on flow analysis that allow to replace PVS non destructive updates with destructive updates.

These analysis require the definition of two versions of each function. A safe version that can always be called and never performs any destructive update and an other "destructive" version that can only be called under certain conditions on the arguments. However thanks to these conditions, the body of that destructive version is allowed to perform safe destructive update and may call destructive versions of functions.

We didn't implement these analysis into the translator but they inspired the static analysis on the intermediate language described next.

3.4 Analysis of the intermediate language

One of the solutions we decided to implement to solve the update problem consists in an analysis on the intermediate language before the optimization and generation of the actual output C code.

We also use two different versions of a PVS function called f and f^d. Our analysis differs from Shankar and Shankar and Cerny's previous work in (at least) the following:

- The non destructive version of a function is also optimized and might have some destructive updates or function calls.
- The destructive function does not have all its function call destructive.

The analysis is explained in details in Section 4. The analysis we implemented basically relies on over approximations of the sets of critical and free variables in a context.

The flags

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 because only A itself points to the newly created array and A is not needed later in the code. We call the following (destructive) 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
...
```

  This flag detects the arguments whose reference can be trapped in the result of a function call. This allows to properly maintain the **mutable** flag.

We want to ensure the following properties on the flags.

safe flag:

- Only a single occurrence of a variable may be flagged **safe** .
- An occurrence of a variable x is flagged **safe** in an instruction iif x occurs once in the instruction and never after.

duplicated flag

- Only expressions and arguments can be flagged **duplicated** .
- If a variable is once flagged **duplicated** , then if it is an argument, this argument is also flagged **duplicated** .

mutable flag:

- Only functions and variables with struct or array (return) types can be flagged **mutable** .
- 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** .

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.

The main difference between the two is that a non destructive function's arguments are never flagged **mutable** . See Listing 2 for an example.

```
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
}
```

Listing 2: Example of the two different versions of a C function generated (stripped from GC instructions)

The algorithm is described Figure 6. It terminates since **duplicated** the flags can only be added, **mutable** flags are removed for good and **safe** flags can only be removed.

- Create the two versions of each function initialized with the same body containing no destructive update or function call.
- For each function:
  - Flag all arguments **mutable** in the destructive version only.
  - Perform several passes **mutable** analysis.
    We flag a variable x when x is initialized with.
    - ⋆ A newly created array (`array(x)`).
    - ⋆ An update (destructive or not).
    - ⋆ A call to a function with a **mutable** return type.

    We remove the **mutable** flag definitely of a variable x when
    - ⋆ x never occurs destructively
    - ⋆ x gets duplicated before its last occurrence

    This allows to make sure the **mutable** properties are verified.
  - Perform a backward pass **duplicated** analysis.
    If x is flagged **duplicated** , then if x is set to...
    - ⋆ ... a function call, flag **duplicated** all variables passed as **duplicated** arguments.
    - ⋆ ... an other expression, flag **duplicated** all the active variables of that expression.
  - Perform a **safe** analysis to flag all variables **safe** if they fulfill the rules described above.
  - Modify the code if the flags allow it according to the rules defined in the Annex D. This may consists in renaming variable, changing the version of a function called,
  - Redo the two previous steps until stabilization.

Figure 6: Algorithm

Link with the analysis

To connect with the analysis Section 4, we could say that

- A occurrence of a variable x is flagged **safe** when this variable is not live in the context of that occurrence.
- The variables flagged **mutable** correspond to the variables that could be critical variables in a context.

  If $f(f_i) := U\{\mathtt{set}(x,\ a);\ e\}$, the variable x is flagged **mutable** when all $Ov(a)$ are flagged **mutable** as well and **safe** . This means all the critical variables of x are neither live in U nor free in e.Then if x is used in a way such that it could get trapped into an other structure (active arguments of a function call, unsafe destructive update, etc) then it loses its **mutable** attribute.
- Arguments f_i are flagged **mutable** , when $f_i \in BA(f)$. This is why we ensure no arguments in non destructive versions of functions are flagged **mutable** .
- An expression a is flagged **duplicated** when $Av(a) \subset Av(e)$. We are only interested in the arguments f_i of f that are flagged **duplicated** though.

3.5 Implemented solution

We decided to implement a reference counting GC that we use to detect on the fly if a variable can be safely updated. However the main optimization is the analysis performed on the intermediate language.

This analysis, however is far from perfect. For instance:

- If a function is called but requires its two argument to be **mutable** and only the first is. Then the non-destructive version is called and the first argument gets copied even though it was **mutable** . We would need probably a lot more versions of functions to solve that problem. However an analysis of all the function calls that are made could narrow the number of versions enough to allow their implementation. This would still increase the size of the code a lot, though.
- We never perform a destructive update on T`[i]` which is never flagged **mutable** (X^* variables are not handled). Our analysis is too simple to tell if T`[i]` is **mutable** or not.

  To prevent that, we also perform GC checks when a safe update occurs. However a more complete implementation of the analysis Section 3.4 could probably solve a lot of these cases.

4 Static analysis of the intermediate language

We describe here the static analysis of the intermediate language (which syntax is defined in Figure 13) implemented in the translator.

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Number} \mid \mathit{Variable} \\
 & \mid & \mathit{Variable}\ \texttt{[}\ \mathit{Variable}\ \texttt{]} \\
 & \mid & \texttt{if (}\ \mathit{Variable}\ \texttt{)}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \\
 & \mid & \texttt{array(}\ \mathit{Variable}\ \texttt{)} \\
 & \mid & \mathit{Variable}\ \texttt{[(}\ \mathit{Variable}\ \texttt{) :=}\ \mathit{Variable}\ \texttt{]} \\
 & \mid & \mathit{Variable}\ \texttt{[(}\ \mathit{Variable}\ \texttt{) <-}\ \mathit{Variable}\ \texttt{]} \\
 & \mid & \texttt{lambda(}\ \mathit{Function}\ \texttt{,}\ \mathit{Number}\ \texttt{,}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{Variable}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{Function}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{PrimOp}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \texttt{set(}\ \mathit{Variable}\ \texttt{,}\ \mathit{Expr}\ \texttt{);}\ \mathit{Expr} \\
\mathit{Variable} & ::= & \mathit{Id} \\
\mathit{Function} & ::= & \mathit{Id} \\
\mathit{PrimOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
 & \mid & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \\
 & \mid & \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{iff} \\
\mathit{FunctionDecl} & ::= & \mathit{Id}\ \texttt{(}\ \mathit{Variable}^*_{,}\ \texttt{)}\ \texttt{=}\ \mathit{Expr} \\
\mathit{Program} & ::= & \mathit{FunctionDecl}^*_{;}\ \mathit{Expr}
\end{array}
$$

Figure 7: Syntax of the intermediate language

We assume a few properties on valid programs:

1. The `set` instruction is a declaration and an assignment of a variable to a value at the same time. If an expression contains two `set` of the same variable, the second `set` will override the first definition. We assume then that a variable is never `set` twice in the same expression.
2. The only free variables in the body of a function declaration are the arguments of that function.

Besides we assume these properties on programs generated by the translator:

1. No destructive update is used to represent PVS updates (before the analysis).

We first define the semantics of the language using a small-steps operational semantics. Then we define a few operators on the language and exhibit some properties. Finally we describe an algorithm to replace non destructive updates with destructive updates under certain conditions and prove that there is a bisimulation between programs before and after applying this algorithm. This proves that the execution of the program is not disturbed by the replacements and thus the correctness of the algorithm.

4.1 Operational semantic

A *value* is either an integer $n \in \mathbb{N}$, a reference $r \in R$ representing an array (or pointer) or a function id $f \in F$. The metavariable v ranges over the set of all values: $V := \mathbb{N} \cup R \cup F$.

An *evaluation context* (sometimes simply called *context*) E is an expression with an occurrence of a hole $[]$. A context and is of one of the forms

1. $[\,]$
2. $\texttt{set(}x\texttt{, }[\,]\texttt{); } e$
3. $\texttt{pop(}[\,]\texttt{)}$
4. $E_1[E_2]$, where E_1 and E_2 are evaluation contexts.

A *redex* is an expression of the following form

1. x
2. $X[y]$
3. $\texttt{if (}x\texttt{) } a \texttt{ else } b$
4. $\texttt{array(}x\texttt{)}$
5. $X\texttt{[(}x\texttt{) := }y\texttt{]}$
6. $X\texttt{[(}x\texttt{) <- }y\texttt{]}$
7. $\texttt{lambda(}f\texttt{, }m\texttt{, }x_1\texttt{, ..., }x_m\texttt{)}$
8. $y\texttt{(}x_1\texttt{, ..., }x_n\texttt{)}$
9. $f\texttt{(}x_1\texttt{, ..., }x_n\texttt{)}$
10. $p\texttt{(}x_1\texttt{, ..., }x_n\texttt{)}$
11. $\texttt{set(}x\texttt{, }v\texttt{); } e$
12. $\texttt{pop(}v\texttt{)}$

We define a *local environment*, s_i, as a function ranging over the set N of all variable names with values in V.The *stack state*, s, is a series of local environments: $s = (s_0, ..., s_n)$.We call $[\,]$ the empty function and if s_i is a local environment ranging over the variables U, we write $s_i \uplus (x \mapsto v)$ the function ranging over $U \cup \{x\}$ mapping x to v and y to $s_i(y)$ for $y \neq x$. For $s = (s_0, ...s_n)$ a stack state, we write $s \uplus (x \mapsto v) := (s_0 \uplus (x \mapsto v), s_1, ..., s_n)$. We also define $s(x)$ as $s_i(x)$ where $\forall j < i, s_j(x)$ is not defined and to simplify notations, we call $s' :: s := (s', s_0, ..., s_n)$ and even $s' :: S := (s' :: s, h)$.

The *heap state* function, h is mapping references r 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, h).We define $S(x) := s(x)$ and $S(r) := h(r)$.

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). A function is associated not only an id but also a number when declared. This number is used in lambda terms to refer to a function using a value.

The meta-variable 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

1. $< x, S > \longrightarrow < S(x), S >$
2. $< x\texttt{[}y\texttt{]}, S > \longrightarrow < h(s(x))(s(y)), S >$
3. $< \texttt{if (}x\texttt{) } a \texttt{ else } b, S > \longrightarrow \begin{cases} < \texttt{pop(}a\texttt{)}, [\,] :: S > & \text{if } s(x) = 0 \\ < \texttt{pop(}b\texttt{)}, [\,] :: S > & \text{otherwise} \end{cases}$
4. $< \texttt{array(}x\texttt{)}, S > \longrightarrow < r, \left(s, h \uplus \left(r \mapsto (0)_{0 \le i < s(x)}\right)\right) >$ where r is a fresh pointer.
5. $< X\texttt{[(}x\texttt{) := }y\texttt{]}, S > \longrightarrow < r, (s, h') >$ where r is a fresh pointer and
$$h' = h \uplus (r \mapsto h(s(X)) \uplus (s(x) \mapsto s(y)))$$
6. $< X\texttt{[(}x\texttt{) <- }y\texttt{]}, S > \longrightarrow < X, (s, h') >$ where
$$h' = h \uplus (s(X) \mapsto h(s(X)) \uplus (s(x) \mapsto s(y)))$$
7. $< \texttt{lambda(}f\texttt{, }m\texttt{, }x_1\texttt{, ..., }x_m\texttt{)}, S > \longrightarrow < r, (s, h') >$ where r is a fresh pointer and
$$h' = h \uplus (r \mapsto (f, m, s(x_1), ..., s(x_m)))$$

8. $< y(x_1,\ \dots,\ x_n), S > \longrightarrow < \texttt{pop}([f]), s' :: S >$ where $E = h(s(y))$, $f = E(0)$, $m = E(1)$,

$$s' : \left| \begin{array}{ccl} \{f_1, \dots, f_{m+n}\} & \to & V \\ f_i & \mapsto & E(i+1) \quad \text{for } i \leq m \\ f_{m+i} & \mapsto & s(x_i) \quad \text{for } i \leq n \end{array} \right.$$

9. $< f(x_1, \dots, x_n), S > \longrightarrow < \texttt{pop}([f]), \left(\biguplus_{1 \leq i \leq n} f_i \mapsto s(x_i)\right) :: S >$
10. $< p(x,\ y), S > \longrightarrow < p(s(x), s(y)), S >$ for binary operators.
11. $< p(x), S > \longrightarrow < p(s(x)), S >$ for the unary operator ($\texttt{not}$).
12. $< \texttt{set}(x,\ v);\ e, S > \longrightarrow < \texttt{pop}(e), (x \mapsto v) :: S >$
13. $< \texttt{pop}(v), ((s_0, \dots, s_n), h) > \longrightarrow < v, ((s_1, \dots, s_n), h) >$

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.

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

One of the greatest advantage of using evaluation contexts is that we define the semantics of this language using only this one small-step rule.

The reflexive-transitive closure of $\longrightarrow$ is represented $\xrightarrow{*}$. The computation of a program is defined as the evaluation of its expression $< e, S_0 >$ on an empty store: $S_0 := (([]), [])$. If $< e, S_0) > \xrightarrow{*} < e', S' >$ then we can prove that $e' \in V$ and the result of the computation is then defined as $eval_{h'}(e')$ where $eval_h$ is defined as follow:

$$eval_h : \left| \begin{array}{lll} V & \longrightarrow & E \\ n & \mapsto & n \in \mathbb{N} \\ r & \mapsto & (eval_h(u_i))_{0 \leq i \leq n} \text{ with } (u_i)_{0 \leq i < n} := h(r) \end{array} \right.$$

Theorem 1. *For all $< e, S_0 > \xrightarrow{*} < v, (s, h) >$, h is defined on $R \cap (\{v\} \cup Im(s) \cup Im(h))$. All references stored in the stack or in the heap or reduced from an expression are defined in the heap state.*

Proof. We proof this theorem by induction on the structure of the code.

1. Base cases: All redexes generating a fresh pointer (4, 5 and 7) modify the heap state to define it on that new pointer.

2. Induction step:

   - References defined in the store are never undefined. If $< e, S > \longrightarrow < e', S' >$ and h is defined on r then h' is also defined on r. This is easily proven since no redex remove a definition in the heap.

   - Whenever a value in the heap or in the store is modified, it is replaced with functions (redex 7), integers (redexes 4 and 7), values from the heap (redexes 2, 5, 6 and 8), values from the stack (redexes 1, 5, 6, 7, 8, 9, 10 and 11) or values reduced from an expression (redexes 12 and 13).

□

4.2 Sets of variables

We define the free variables, Fv, of an expression as the set of all variables that occur in that expression. For this study, we are only interested in variables that may refer to an array.

We also define the *output* variables, $Ov(e)$, of an expression. This can be understood in three ways:

- This is the set of all variables that may have their value "trapped" into the expression e.
- It corresponds to all the variables which content might get modified if e gets modified in place.
- It is the set of all variables which content the pointer corresponding to the evaluation of e may point to.

Finally, we define the *active* variables, Av, of an expression. In a similar way, this can be understood as:

- The set of all variables that may contain a reference to the expression e.
- It corresponds to all the variables that could be accessed from the value returned by the expression.
- It is the set of all variables which evaluation may be a pointer pointing to e.

If X s a variable, X^* refers to all the references X may point to. It is obvious that if $X \in Av(e)$, then $X^* \in Av(e)$ as well. In that case, we voluntarily omit to mention X^* in the definition of these set below.

Expression	Ov	Av	Fv
n	$\emptyset$	$\emptyset$	$\emptyset$
X	$\{X\}$	$\{X\}$	$\{X\}$
X`[`x`]`	$\{X^*\}$	$\{X^*\}$	$\{X\}$
`if (`x`)` a `else` b	$Ov(a) \cup Ov(b)$	$Av(a) \cup Av(b)$	$Fv(a) \cup Fv(b)$
`array(`x`)`	$\emptyset$	$\emptyset$	$\emptyset$
X`[(`x`) :=` y`]`	$\emptyset$	$\{X^*, y\}$	$\{X, y\}$
X`[(`x`) <-` y`]`	$\{X\}$	$\{X, y\}$	$\{X, y\}$
`lambda(`f`,` m`,` x_1`, ...,` x_m`)`	$\emptyset$	$\{x_1, .., x_m\}$	$\{x_1, .., x_m\}$
y`(`x_1`, ...,` x_n`)`	$\{x_1, x_1^*, ..., x_n, x_n^*\}$	$\{x_1, ..., x_n\}$	$\{y, x_1, .., x_n\}$
f`(`x_1`, ...,` x_n`)`	$\{x_i \mid f_i \in Ov([f])\}$ $\cup \ \{x_i^* \mid f_i^* \in Ov([f])\}$	$\{x_i \mid f_i \in Av([f])\}$ $\cup \ \{x_i^* \mid f_i^* \in Av([f])\}$	$\{x_1, .., x_n\}$
p`(`x_1`, ...,` x_n`)`	$\emptyset$	$\emptyset$	$\emptyset$
`set(`x`,` a`);` e	$Ov(e) \cup Av(a) - \{x, x^*\}$ if $x^* \in Ov(e)$ $Ov(e) \cup Ov(a) - \{x\}$ if $x \in Ov(e)$ $Ov(e)$ otherwise	$Av(e) \cup Av(a) - \{x, x^*\}$ if x or $x^* \in Av(e)$ $Av(e)$ otherwise	$Fv(a) \cup Fv(e) - \{x\}$

Theorem 2. *For all expression e, $Ov(e) \subset Av(e) \subset Fv(e)$.*

Proof. Simple induction proof on the expression form. □

4.3 Update contexts

The advantage of using contexts is to be able to place critical expressions like updates into a context where the evaluation order is well defined and we can identify expression evaluated before and after the reduction of a critical redex like a function call or a destructive update. These two being the only redexes that can modify the heap store in place.

We introduce *update contexts* as an expression with a single occurrence of a hole:

1. $\{\}$
2. $\texttt{set(}x\texttt{, }U\texttt{); }e$
3. $\texttt{set(}x\texttt{, }a\texttt{); }U$
4. $\texttt{if (}x\texttt{) }U\texttt{ else }b$
5. $\texttt{if (}x\texttt{) }a\texttt{ else }U$

To define which update can be made destructively, we now build a reference graph in a context U. We define the pointer analysis of U as $PA(U)$ where $PA(U)(X)$ is the set of all variables X may point to in the context U. It is the smallest set containing $PA^0(U)(X)$ and close under $x \in PA(U)(X) \Longrightarrow PA(U)(x) \subset PA(U)(X)$.

$$\begin{array}{rcl}
PA^0(\{\}) & : & \left| \begin{array}{lll} X & \mapsto & \{X, X^*\} \\ X^* & \mapsto & \{X^*\} \end{array} \right. \\
PA^0(\texttt{set(}x\texttt{, }U\texttt{); }e) & := & PA(U) \\
PA^0(\texttt{set(}x\texttt{, }a\texttt{); }U)(X) & := & PA^0(U)(X) \cup \left\{ \begin{array}{ll} \{x, x^*\} & \text{if } X \in Ov(a). \\ \{x^*\} & \text{if } X \in Av(a). \\ Av(a) & \text{if } x^* \in PA^0(U)(X). \\ \emptyset & \text{otherwise.} \end{array} \right. \\
PA^0(\texttt{if (}x\texttt{) }U\texttt{ else }b) & := & PA^0(U) \\
PA^0(\texttt{if (}x\texttt{) }a\texttt{ else }U) & := & PA^0(U)
\end{array}$$

This will allow us to define the variables live in this context as well as the set of critical variables that could be modified by a destructive update.

- For X a variable and U a context, the set of critical variables $Cv(U)(X)$ contains all variables that may point to X and all variable that may point to these variables and so on.

$$Cv(U)(X) := \{ \ y \mid X \in PA(U)(y) \ \} = PA(u)^{-1}(\{X\})$$

- The set $Lv(U)$ of the variables live in the context U are the variables that could be evaluated after the hole the in the context and the variables that these variables may point to and so on.

$$\begin{array}{rcl}
Lv^0(\{\}) & := & \emptyset \\
Lv^0(\texttt{set(}x\texttt{, }U\texttt{); }e) & := & Fv(e) \cup Lv^0(U) \\
Lv^0(\texttt{set(}x\texttt{, }a\texttt{); }U)(X) & := & Lv^0(U) \\
Lv^0(\texttt{if (}x\texttt{) }U\texttt{ else }b) & := & Lv^0(U) \\
Lv^0(\texttt{if (}x\texttt{) }a\texttt{ else }U) & := & Lv^0(U) \\
Lv(U) & := & \displaystyle\bigcup_{x \in Lv^0(U)} PA(U)(x)
\end{array}$$

When we consider turning a non destructive update $X\texttt{[(}x\texttt{) := y]}$ in a context U into a destructive update, we want to make sure that critical variables are not live U.

$$Cv(U)(X) \cap Lv(U) = \emptyset$$

4.4 Analysis

We consider here a function f which declaration body e doesn't contain any destructive update before the analysis. The reason is that a safe, naive translation from PVS to this language would only perform non destructive updates.

Intuitively, if a function $f(f_1, \dots, f_n) = e$ contains a non destructive update in a context $e = U\{X[(x) := y]\}$. That update can be turned into a destructive update if none of the variables that may be aliased to X are live in the context.

$$Cv(U)(X) \cap Lv(U) = \emptyset$$

This way all this variables that may point to X are never used after the update. Since they are the only variables whose evaluation is modified by making the update destructive, we can say that this is a safe transformation.

The problem is that some of these variables possibly pointing to X might be included in the set of arguments of f: $\{f_1, ..., f_n\}$. And we can't assume anything about these variables since we don't have any information regarding the context in which the function f is called.

We define the bang analysis $BA(f)$ as the set of all variables that are involved in a destructive update or are destructive arguments in a call to a destructive function:

$$BA(f) \;:=\; \left(\bigcup_{e=U\{X[(x)\ \texttt{<-}\ y]\}} Cv(U)(X)\right) \cup \left(\bigcup_{e=U\{g(x_1,\dots,x_n)\}} Cv(U)(\{x_i | g_i \in BA(g)\})\right)$$

Basically $BA(f)$ is the set of all critical variables in the evaluation of a function call to f.

We define two versions of all function declared. A non destructive version f with body e_{nd} and a destructive version f^d with body e_d. Both new definitions may only differ from the original body e of f in some very specific substitutions: non destructive updates in e may be destructive in e_{nd} and e_d and function calls to a function g may become function calls to g^d with the same arguments in e_d and e_{nd}.

These functions use two different strategies:

- The second consists in allowing destructive updates of variables that could be aliased arguments. We however keep track of these arguments and only call this function when we are sure the arguments are safe in the context of the call. The yields the definition e_d of f^d verifying the following properties:
  - If $e = U\{X[(x)\ \texttt{:=}\ y]\}$ and U' is the corresponding context in e_d then

    $$e_d = U'\{X[(x)\ \texttt{<-}\ y]\} \iff Cv(U')(X) \cap Lv(U') = \emptyset$$

  - If $e = U\{g^d(x_1,\dots,x_n)\}$ and U' is the corresponding context in e_d then the condition for $e_d = U'\{X[(x)\ \texttt{<-}\ y]\}$ is

    All $Cv(U')(x_i)$ for $g_i^d \in BA(g^d)$ and $Lv(U')$ are pairwise disjoints.

- The first consists in forbidding the use of destructive updates which argument may be pointing to by arguments. This is equivalent to saying that all arguments of non destructive functions are live in all contexts (or just in the empty context). This yields the new definition e_{nd} of f verifying the following properties:
  - If $e = U\{X[(x)\ \texttt{:=}\ y]\}$ and U' is the corresponding context in e_{nd} then

    $$e_{nd} = U'\{X[(x)\ \texttt{<-}\ y]\} \iff Cv(U')(X) \cap (Lv(U') \cup \{f_1, ..., f_n\}) = \emptyset$$

- If $e = U\{\texttt{g}^d\texttt{(}x_1\texttt{,...,}x_n\texttt{)}\}$ and U' is the corresponding context in e_{nd} then the condition for $e_d = U'\{X\texttt{[(}x\texttt{) <- }y\texttt{]}\}$ is

$$\text{All } Cv(U)(x_i) \text{ for } g_i^d \in BA(g^d) \text{ and } Lv(U) \cup \{f_1, ..., f_n\} \text{ pairwise disjoints.}$$

Proposition 3. *For all non destructive version of a function f, $BA(f) = \emptyset$.*

4.5 Proof of bisimulation

We define the accessible cells, from a variable X given a store S as the set of all references (heap store entries) that can be accessed from the variable X.

$$Ac(S)(X) := R \cap S^{\infty}(X)$$

We extend that definition of S to star variables and of Ac to expressions and updates

$$S(X^*) := Ac(S)(X)$$
$$\begin{aligned} Ac(S)(e) &:= Ac(S)(Av(e)) \\ Ac(S)(U) &:= Ac(S)(Lv(U)) \end{aligned}$$

It correspond to the only references already created in the heap store that can still be accessed in the context or expression.

The proof of correctness relies on three main invariants:

1. The function calls to f or f^d have the same evaluation. This means they can't be told apart by looking only at the value of the result. However they differ in the use of their arguments.

2. When the function call $f(x_i)$ is evaluated with a store S, the only references which value could be modified is $\bigcup_{f_i^{(*)} \in BA(f)} S\left(x_i^{(*)}\right)$.

3. The union above is always a pairwise disjoint union.

These properties are obviously true before the analysis since $f = f^d$, $BA(f) = \emptyset$ for all function and no reference ever have its value modified.

The destructive and non destructive versions of a function f are built from the original definition e by replacing function calls (updates can be considered as a function call). To prove the correctness of this algorithm we need to prove that the three invariants described above are preserved between programs before and after a single replacement.

We consider the replacement of g to g^d in a function f^d of body $e = U\{\texttt{g}^d\texttt{(}x_i\texttt{,}y_j\texttt{,}z_k\texttt{)}\}$ with x_i variables corresponding to the critical arguments $g_i^d \in BA(g^d)$, y_j corresponding to arguments that are active in the body of e and z_k arguments that are not arrays or are not in either of the previous sets.

We have the following hypothesis

($H1$) All $Cv(U)(x_i)$ and $Lv(U)$ are pairwise disjoints (Analysis).

($H2$) The evaluation of the call to g^d with a store S is the same than if g was called.

($H3$) The only references which value may be modified during the evaluation of the call with a store S is the union $\bigcup_{f_i^{(*)} \in BA(f)} S\left(x_i^{(*)}\right)$.

($H4$) f will always be called in an evaluation context with a store S such that the sets $S\left(f_i^{(*)}\right)$ are disjoint for all $f_i^{(*)} \in BA(f)$.

Proposition 4. *For all store S with which this function call is evaluated, the sets $S\left(x_i^{(*)}\right)$ are pairwise disjoint.*

Proof. $(H1)$ and $(H4)$. $\square$

Proposition 5. *For all store S with which the function call is evaluated, the sets $\bigcup S\left(x_i^{(*)}\right)$ and $Ac(S)(U)$ are disjoints.*

Proof. $(H1)$ and $(H4)$. $\square$

Proposition 6. *When a call to f^d is evaluated with a store S the only references that may be modified in S is $\bigcup_{f_i^{d,(*)} \in BA(f)} S\left(x_i^{(*)}\right)$*

Proof. By definition of $BA(f)$, all the arguments that may point to references that could newly be modified are added to $BA(f)$. $\square$

Proposition 7. *The function calls to f or f^d have the same evaluation.*

Proof. The only modification in the evaluation of a function call to the new f^d is the function call to g^d. That function call has the same evaluation, $(H2)$, and the only references it may modify, $(H3)$ are never used afterward in the evaluation of the body of f^d, $(H1)$. $\square$

This analysis would have worked as well for non destructive versions of functions and for the replacement of an update rather than a function call.

Theorem 8. *If $< e_1, (s_1, h_1) > \longrightarrow < e_2, (s_2, h_2) >$ then $< e_1, (s_1, h_1') > \longrightarrow < e_2, (s_2, h_2') >$ where $h_i' = h_i|_{Lc(S_1)(e_1)}$.*

Proof. It can easily be verified for every redex. When $a = U\{b\}$, we have $Lc(a) = Lc(U) \cup Lc(b)$ and the inductive step is proved . $\square$

5 Conclusion

We have described the general architecture of the translator from PVS to the C language we implemented and integrated into PVS. We have provided a few examples of its execution on simple examples to illustrate its mechanisms.

We have described the update issue and various ways to deal with it. Both the use of a reference counting garbage collector and an analysis of the intermediate language were implemented.

We have defined the semantics of the intermediate language which allowed us to prove the correctness of the analysis that is performed on it to eliminate non destructive updates.

This tool allows to efficiently execute PVS code for debugging and testing purposes and to easily integrate C code into actual systems where the C and C++ are common development languages.

5.1 Difficulties and successes

This project was a great challenge and an opportunity for me to conduct my own autonomous research on a subject I chose. The development of the working translator was also the occasion for me to discover new tools, formal techniques and learn a lot about computer science in general.

Working with new languages and tools

To be able to translate PVS, I had to fully understand not only the syntax and semantics of the PVS language but also the structure of the PVS API written in Common Lisp. This means I also had to learn Common Lisp which I decided then to use to write the translator mostly because it made the integration of the native PVS parser and typechecker easier. Finally I had to discover the C language which I only had a basic knowledge of.

Integrating the GMP library

In PVS (and in other languages such as Common Lisp or Python), the `integer` type represent the whole set $\mathbb{Z}$ of all relative numbers (and `rational` also describes the whole set $\mathbb{Q}$). To implement that in C, we need more than the finite types `int`, `long`, etc.

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

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

Figure 8: Example of the GMP library use

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

For example (Figure 8), 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`.

5.2 What's left to be done ?

One of my biggest regret was not having the time to finish the translator and properly implement closures. Some work is already done in that direction though. It relies on a C structure to represent a closure:

```
struct r_list_int {
   int (*body)(void* env, void* args);    // body is a function pointer
   void* env;                              // env contains the environment variables
   void* args;                             // args will contain the arguments
};
```

Since the translator's correctness itself has not be proven, there is no formal guarantee that the semantic will be preserved during the translation. In particular, even if some properties were proven on a certain PVS model, its translation to C could not verify these properties. This tool certainly does not allow the generation of high insurance code. The only way to do that would be to formally prove the correctness of the translator. The CompCert C compiler (http://compcert.inria.fr/compcert-C.html), is an example of such a proven translator.

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

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

5.3 My stay at SRI

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

The first weeks of my stay were the occasion to discover PVS and 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 discovering the middle- and back-end of the PVS API. Among other excerises, I decided to write a Common Lisp parser to help me understand the huge architecture of the PVS API 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 "Crazy Ideas" seminar hosted every other week was a ...

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

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

References

[ČS06] Pavol Černý and N. Shankar. "Static Analyses for Guarded Optimizations of High Level Languages". SRI Internship Report. MA thesis. ENS Paris, 2006 (cit. on pp. 6, 16).

[Fil14] Jean-Christophe Filliâtre. *Langages de programmation et compilation*. MPRI. https://www.lri.fr/~filliatr/ens/compil/. 2013-14.

[Hus04] Eric Huss. “The C Library Reference Guide”. In: *Webmonkeys: A Special Interest* (2004) (cit. on p. 7).

[JL] Richard Jones and Rafael Lins. “Garbage Collection: Algorithms for Automatic Dynamic Memory Management, 1996”. In: *John Wiliey & Sons Ltd., England* () (cit. on p. 11).

[Owr+98] S. Owre et al. *User Guide for the PVS Specification and Verification System*. Three volumes: Language, System, and Prover Reference Manuals. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1998 (cit. on p. 6).

[Owr+99a] S. Owre et al. *PVS Language Reference*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1999 (cit. on p. 6).

[Owr+99b] S. Owre et al. *PVS System Guide*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1999 (cit. on p. 6).

[Sha02] Natarajan Shankar. “Static analysis for safe destructive updates in a functional language”. In: *Logic Based Program Synthesis and Transformation*. Springer, 2002, pp. 1–24 (cit. on pp. 6, 16).

[SO03] N. Shankar and S. Owre. *PVS API Reference*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 2003 (cit. on p. 6).

A PVS syntax and CLOS representation

$$
\begin{array}{lll}
\mathit{Expr} & ::= & \mathit{Number} \mid \mathit{Name} \mid \mathit{Expr}\ \texttt{(}\ \mathit{Expr}^{+}_{,}\ \texttt{)} \\
 & \mid & \mathit{Expr}\ \mathit{Binop}\ \mathit{Expr} \\
 & \mid & \mathit{Unaryop}\ \mathit{Expr} \\
 & \mid & \mathit{Expr}\ \texttt{`}\ \{\ \mathit{Id} \mid \mathit{Number}\ \} \\
 & \mid & \texttt{(}\ \mathit{Expr}^{+}_{,}\ \texttt{)} \\
 & \mid & \texttt{(\#}\ \mathit{Assignment}^{+}_{,}\ \texttt{\#)} \\
 & \mid & \mathit{IfExpr} \\
 & \mid & \texttt{LET}\ \mathit{LetBinding}^{+}_{,}\ \texttt{IN}\ \mathit{Expr} \\
 & \mid & \mathit{Expr}\ \texttt{WHERE}\ \mathit{LetBinding}^{+}_{,} \\
 & \mid & \mathit{Expr}\ \texttt{WITH [}\ \mathit{Assignment}^{+}_{,}\ \texttt{]} \\
\mathit{Number} & ::= & \mathit{Digit}^{+} \\
\mathit{Id} & ::= & \mathit{Letter}\ \mathit{IdChar}^{+} \\
\mathit{IdChar} & ::= & \mathit{Letter} \mid \mathit{Digit} \\
\mathit{Letter} & ::= & \texttt{A} \mid \ldots \mid \texttt{Z} \\
\mathit{Digit} & ::= & \texttt{0} \mid \ldots \mid \texttt{9} \\
\mathit{IfExpr} & ::= & \texttt{IF}\ \mathit{Expr}\ \texttt{THEN}\ \mathit{Expr} \\
 & & \{\texttt{ELSIF}\ \mathit{Expr}\ \texttt{THEN}\ \mathit{Expr}\}^{*}\ \texttt{ELSE}\ \mathit{Expr}\ \texttt{ENDIF} \\
\mathit{Name} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{integer?} \\
 & \mid & \texttt{floor} \mid \texttt{ceiling} \mid \texttt{rem} \mid \texttt{ndiv} \mid \texttt{even?} \mid \texttt{odd?} \\
 & \mid & \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{cons?} \mid \texttt{null} \mid \texttt{null?} \\
 & \mid & \texttt{restrict} \mid \texttt{length} \mid \texttt{member} \mid \texttt{nth} \mid \texttt{append} \mid \texttt{reverse} \\
\mathit{Binop} & ::= & \texttt{=} \mid \texttt{\textbackslash=} \mid \texttt{OR} \mid \texttt{\textbackslash/} \mid \texttt{AND} \mid \texttt{\&} \mid \texttt{/\textbackslash} \\
 & \mid & \texttt{IMPLIES} \mid \texttt{=>} \mid \texttt{WHEN} \mid \texttt{IFF} \mid \texttt{<=>} \\
 & \mid & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{Unaryop} & ::= & \texttt{NOT} \mid \texttt{-} \\
\mathit{Assignment} & ::= & \mathit{AssignArg}^{+}\ \{\texttt{:=} \mid \texttt{|->}\}\ \mathit{Expr} \\
\mathit{AssignArg} & ::= & \texttt{(}\ \mathit{Expr}^{+}_{,}\ \texttt{)} \mid \texttt{`}\ \mathit{Id} \mid \texttt{`}\ \mathit{Number} \\
\mathit{LetBinding} & ::= & \{\mathit{LetBind} \mid \texttt{(}\ \mathit{LetBind}^{+}_{,}\ \texttt{)}\}\ \texttt{=}\ \mathit{Expr} \\
\mathit{LetBind} & ::= & \mathit{Id}\ [\texttt{:}\ \mathit{TypeExpr}]
\end{array}
$$

Figure 9: 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 10: (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 11: Fragment of the PVS type system

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

Figure 12: (Partial) CLOS representation of PVS types

C Intermediate languages

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Number} \mid \mathit{Variable} \\
 & \mid & \mathit{Variable}\ \texttt{[}\ \mathit{Variable}\ \texttt{]} \\
 & \mid & \texttt{if (}\ \mathit{Variable}\ \texttt{)}\ \mathit{Expr}\ \ \texttt{else}\ \ \mathit{Expr} \\
 & \mid & \texttt{array(}\ \mathit{Variable}\ \texttt{)} \\
 & \mid & \mathit{Variable}\ \texttt{[(}\ \mathit{Variable}\ \texttt{) :=}\ \ \mathit{Variable}\ \texttt{]} \\
 & \mid & \mathit{Variable}\ \texttt{[(}\ \mathit{Variable}\ \texttt{) <-}\ \ \mathit{Variable}\ \texttt{]} \\
 & \mid & \texttt{lambda(}\ \mathit{Function}\ \texttt{,}\ \ \mathit{Number}\ \texttt{,}\ \ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{Variable}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{Function}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \mathit{PrimOp}\ \texttt{(}\ \mathit{Variables}\ \texttt{)} \\
 & \mid & \texttt{set(}\ \mathit{Variable}\ \texttt{,}\ \ \mathit{Expr}\ \ \texttt{);}\ \ \mathit{Expr} \\
\mathit{Variable} & ::= & \mathit{Id} \\
\mathit{Function} & ::= & \mathit{Id} \\
\mathit{PrimOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
 & \mid & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \\
 & \mid & \texttt{not} \mid \texttt{and} \mid \texttt{or} \mid \texttt{iff} \\
\mathit{FunctionDecl} & ::= & \mathit{Id}\ \texttt{(}\ \ \mathit{Variable}^{*}_{,}\ \ \texttt{)}\ \ \texttt{=}\ \ \mathit{Expr} \\
\mathit{Program} & ::= & \mathit{FunctionDecl}^{*}_{;}\ \mathit{Expr}
\end{array}
$$

Figure 13: Syntax of the intermediate language

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

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

Variable      ::=  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 14: Syntax of the representation language. Every *Expr* is typed.

D Rules

	A safe	**A not safe**

A mutable	`A[k] = v;`
	Replace every occurrence of the variable `B` by the variable `A`
		`B = GC_malloc(...);`
		`for(i ...)`
		`  B[i] = A[i];`
		`B[k] = v;`

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

Figure 15: Rules for `set(B, A[(i) := v])`

	A safe	**A not safe**

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

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

Figure 16: 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 17: Rules for `f(A)` with `A` flagged **mutable** in the destructive version

E Examples

Here are a few simple example to get an idea of the optimization that occur on the intermediate language. Following is a complete example of a program generating an array of pseudo random numbers and sorting it with the insertion sort algorithm.

PVS code	Intermediate language code (before analysis - with types)	C code generated (after analysis)

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

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

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

Figure 18: Examples of setting variables

PVS code	Intermediate language code	C code generated

`f(A:Arr):Arr =`
`  A WITH [(0) := 0]`
	`f:  ( int* A ) -> int*`
	`set(L, 0);`
	`set(R, 0);`
	`A[(L) := R]`
	
	(after analysis)
	
	`fd:  ( int* A ) -> int*`
	`set(L, 0);`
	`set(R, 0);`
	`A[(L) <- R]`
		`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;`
		`}`

`f(A:Arr):Arr =`
` let B = A WITH[(0):=0]`
` in A WITH[(0) := B(0)]`
	`f:  ( int* A ) -> int*`
	`set(L1, 0);`
	`set(R1, 0);`
	`set(B, A[(L1) := R1] );`
	`set(L2, 0);`
	`set(R2, B(0));`
	`A[(L2) := R2] );`
	
	(after analysis)
	
	`fd:  ( int* A ) -> int*`
	`set(L1, 0);`
	`set(R1, 0);`
	`set(B, A[(L1) := R1] );`
	`set(L2, 0);`
	`set(R2, B(0));`
	`A[(L2) <- R2] );`
		`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;`
		`}`

Figure 19: Examples of copying variables

```
benchmark : THEORY
BEGIN

% We use the Lehmer random number generator
% with the following parameters

% n      = 59557   big prime number picked from
%                  http://primes.utm.edu/lists/small/10000.txt
% length = 1000
% g      = 12345
% X_0    = 9876

  SIZE: int = 1000

  Val  : TYPE+ = subrange(0, 59557)
  Ind  : TYPE+ = below(SIZE)
  Arr  : TYPE+ = [ Ind -> Val ]

  A : VAR Arr
  i : VAR Ind
  v : VAR Val

  init(A, i, v): RECURSIVE Arr =
    let B = A with [(i) := v] in
      if i >= SIZE-1 then B
      else init(B, i+1, rem(59557)(12345 * v) ) endif
  MEASURE SIZE - 1 - i

  J :Arr = lambda(k:Ind): SIZE - 1 - k
  Z :Arr = lambda(x:Ind) : 0
  T :Arr = init(Z, 0, 9876)


  insert(A, v, i): RECURSIVE Arr =
    IF (i = 0 OR v >= A(i - 1))
    THEN A WITH [(i) := v]
    ELSE insert(A WITH [(i) := A(i - 1)], v, i - 1)
    ENDIF
    MEASURE i

  insort_rec(A, (n:upto(SIZE)) ): RECURSIVE Arr =
    IF n < SIZE THEN
      let An = A(n) in
        insort_rec( insert(A, An, n), n + 1 )
    ELSE A ENDIF
    MEASURE SIZE - n

  insort(A): Arr = insort_rec(A, 0)

  tsort: Val = insort(T)(0)
  jsort: Val = insort(J)(0)

END benchmark
```

Figure 20: Full PVS example - PVS theory

```c
#define SIZE 1000;
#define SIZE_1 SIZE-1;

unsigned long int* init(unsigned long int* A, int i, unsigned long int v) {
  A[i] = v;
  if ((i >= SIZE_1))
    return A;
  else
    return init( A , (i + 1) , ((12345 * v) % 59557) );
}

unsigned long int J(int k) {
  return (unsigned long int) (SIZE_1 - k);  }

unsigned long int Z(int x) {
  return (unsigned long int) 0;  }

unsigned long int* T() {
  unsigned long int* aux = GC_malloc(SIZE, sizeof(unsigned long int) );
  int i;
  for(i = 0; i < SIZE; i++)
    aux[i] = Z( i );
  return init( aux , 0 , (unsigned long int) 9876 );
}

unsigned long int* insert(unsigned long int* A, unsigned long int v, int i) {
  if (((i == 0) || (v >= A[(i - 1)]))) {
    A[i] = v;
    return A;
  } else {
    unsigned long int res = A[(i - 1)];
    A[i] = res;
    return insert( A , v , (i - 1) );
  }
}

unsigned long int* insort_rec(unsigned long int* A, int n) {
  if ((n < SIZE)) {
    unsigned long int An = A[n];
    return insort_rec( insert( A , An , n ) , (n + 1) );
  } else  return A;
}

unsigned long int* insort(unsigned long int* A) { return insort_rec( A , 0 ); }

unsigned long int tsort() { return insort( T() )[0]; }

unsigned long int jsort() {
  unsigned long int* aux;
  aux = GC_malloc(SIZE, sizeof(unsigned long int) );
  int i;
  for(i = 0; i < SIZE; i++)
    aux[i] = J( i );
  return insort( aux )[0];
}
```

Figure 21: Full PVS example - C translation