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

June 23, 2014

Contents

1 Introduction

2 PVS

3 Translating PVS to C

4 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.

5 PVS Syntax

We describe here the syntax of PVS and the objects system used to represent them in Lisp. Some slots of the classes are voluntarily omitted. For a full description of PVS parser representation, refer to [2].

```
Expr        ::=  Number
              |  Name
              |  Expr Arguments
              |  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

Arguments   ::=  ( Expr+, )

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

Name        ::=  true | false | number_field_pred | real_pred
              |  integer_pred | integer? | rational_pred
              |  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]
```

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]*

number a nonnegative integer

`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

actuals a list of actuals

argument the argument

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

`record-expr` ⊂ `expr` *[class]*

assignments a list of assignments

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

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

`if-expr` ⊂ `application` *[class]*

When an application has an operator that resolves to the `if_def` it is changed to this class.

`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 `assignment` instances. Note that these are very succinct representations, but correspondingly difficult to typecheck or to translate to other systems (e.g., decision procedures). See the description of the function `translate-update-to-if!` for more details.

`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`.

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);]
```

6 Types

A PVS theory can be typechecked using the emacs interface `M-x typecheck` or with 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. The syntax of PVS we allow

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*

`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

. . . .

7 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

Auxiliary 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 auxiliary language :

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

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

C types:[unsigned] char, int, long, double boolean arrays strings enum struct and others: short int, float, union, size_t, ...

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

7.1 Translating PVS syntax

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

7.2 Difficulties

7.2.1 `if` expressions

Represented by `if-expr`

7.2.2 Integer, rationnals

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

We need the GMP library which introduces the types `mpz_t`and `mpq_t`. These types are arrays and should be used just as integer (not as pointers except they still need to be freed).

7.2.3 Garbage collection

We implement a very simple "Reference Counting Garbage Collector" as described in [1].

We maintain a hashtable of pointer counters. Each pointer in the code is a key in the hastable to which we associate an int counter as value.

Pointers only occurs in arrays or struct.

Arrays are created in the code.

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

becomes

```
T* a = GC( b ); // Should not happen often...
```

```
t[0] = b;  // with b of type T*
```

becomes

```
GC_free( t[0] );
t[0] = (T*) GC( b );
```

Examples

```
int* f() {
  int* res;
  res = (int*) GC_alloc( 10 * sizeof(int) );
  [... init res...]
  return res; // pointer count = 1
}

void main() {
   int* a = f();  // pointer counter of a = 1
   int** b = (int**) GC_alloc( sizeof( int*) );
   GC_free( b[0] );  // useless
   b[0] = (int*) GC( a );  // pointer counter of a = 2
   printf("f(0)␣=␣%s", b[0][0]);
   GC_free(b); // frees b, pointer count of a = 1
   GC_free(a); // frees a
}
```

7.2.4 Update expressions

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`.

Three solutions :

- Pointer counting :

We keep track of the number of pointer pointing to an array or a struct.

This requires to build our own C struct (heavy)

```
struct array_int {
   int pointer_count = 1;
```

```
    int *data;
};
```

When we update the struct, if the pointer is 0, we update in place.

Besides every update require now to read the structure and make a test (small compared to a copy but no so small compared to a single in place update)

Besides, the creation / destruction gets more complicated

Passing argument to function :

```
array_int f(array_int arg) {
    arg.pointer_count++;  \\ Since now f also have a pointer to the struct
    if (arg.pointer_count == 1) {
        arg.data[0] = 0;
        return arg;
    } else {
        array_int res;
        res.data = malloc( 10 * sizeof(int*) );
        copy(res, arg);  // Very long...
        res.pointer_count --; // This function is about to lose its pointer to re
        return res;
    }
}

void main() {
    array_int t;
    init(t);  // somehow...

    t.pointer_counter --;  // We assure we won't use the pointer "t" to the arra
    array_int r = f(t);
    t = null; // This way we garantee the variable "t" won't be used later in th

    [...]
}
```

This add quite some code compared to the simple :

```
array_int f(array_int arg) {
    arg[0] = 0;
    return arg;
}

void main() {
    array_int t;
    init(t);  // somehow...
    array_int r = f(t);

    [...]
}
```

- Using a different data structure

PVS uses arrays in a very particular way, we might then represent them with an other structure than just only a C array. For example :

```
struct r_list_int {
    int k;
    int v;
    r_list_int tl;
};

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

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

Problems : Just as the previous solution : - add some extra code - add some extra computation (runtime tests) - require to create as many structures and associated functions as there are range type fo the manipulated arrays

- Third solution :

Trying to avoid copying arrays by analyzing the code. 2 different functions (destructive and non destructive)

Algorithm :

Always have a "non destructive" version of any function. A "cautious" version that never modify the arguments in place and always make copies when necessary (when a "mutable" version of an array is necessary (for instance updates)).

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 observe 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

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(g(t), t) if g has type [Array! -¿ ?] then t can't be destructive if g has type [Array -¿ ?] then t can be destructive

need multiple passes as the flags disappear

8 Other works at SRI

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

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

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