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 19, 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 **?**.

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*]

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 add:

$\mathrm{d}_a dd(*mpz_t res, mpz_t[mutable]a, long b) mpz_a dd(a, a, b); (*res) = a; Rq : d_a dd is given a mutable mpz_t, meanir$

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

if-expr update-expr

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