The Coq proof assistant - second part

Verification of a compiler

Catherine Dubois[1]

[1]ENSIIE - CEDRIC, Évry, France

TOOLS 2011

Compiler Verification as a case study

Compilation of exceptions to virtual machine code

- Minimal source language: arithmetic expressions + throw an exception + catch an exception.
- Basic method of compiling using stack unwinding

Mechanisation of a proof done manually by G. Hutton and J. Wright (Nottingham) (also mechanised by T. Nipkow using Isabelle, some years ago).

Ref: Graham Hutton, Joel Wright: Compiling Exceptions Correctly. MPC 2004: 211-227

Roadmap

2 steps

- From arithmetic expressions to code for a stack machine (see the file simple_compiler.v)
  - syntax
  - semantics
  - compiler
  - proof of correctness
- Adding exceptions

Arithmetic expressions

Syntax

- A minimal language: numbers and additions
- Deep embedding (versus shallow embedding) in Coq
- We define the type of arithmetic expressions as an inductive type:

```
Inductive  expr : Set :=
   Val : nat -> expr
| Add : expr -> expr -> expr.
```

Definition of an interpreter eval (from expr to nat)

A recursive function in Coq

```
Fixpoint eval (e : expr) : nat :=
match e with
  Val i => i
| Add e1 e2 => eval e1 + eval e2
end.

Eval compute in eval (Add (Val 5) (Add (Val 4) (Val 3))).
12 : nat
```

The virtual machine

Syntax of operations

- A stack of natural numbers:

```
Definition stack := (list nat).
```

- 2 operations:
  - Push a natural number on top
  - Add the two numbers on top

```
Inductive Op : Set :=
   PUSH : nat -> Op
|  ADD : Op.
```

- A code is a list of operations:

```
Definition code := list Op.
```

The virtual machine

Semantics

The execution of a program can be seen as a function of type `code -> stack -> stack`.
But the function is not total: some programs have no semantics, e.g. [PUSH 2;ADD] (stack underflow). :-(

- Ocaml, Java: Throw an exception in such a case

- Coq: 2 possible definitions:

1. as a function that mimics exceptions:
   `code -> stack -> option stack` with

   ```
   Inductive option (A : Type): Type :=
       Some : A -> option A | None : option A
   ```

2. as a relation exec defined inductively:
   `code -> stack -> stack -> Prop`

   `exec ops s s'` : read "the execution of the code ops with the initial stack `s` results in the final stack `s'`".

As we are interested in the semantics to prove the correctness of the compiler, we chose the second solution.

```
Inductive exec : code -> stack -> stack -> Prop :=
  exnil : forall s, exec nil s s
| expush :  forall cs i s f, exec cs (i::s) f ->
    exec (PUSH i)::cs s f
| exadd : forall cs i j s f, exec cs ((j+i)::s) f ->
    exec (ADD::cs) (i :: j :: s) f.
```

Inductive extraction on that predicate with mode [1,2] gives us the expected function.

It' s easy to prove exec is deterministic
(by induction on `(exec ops s s1)` with a heavy use of the `inversion` tactic)

```
Lemma exec_det : forall ops s s1 s2,
exec ops s s1 -> exec ops s s2 -> s1=s2.
```

```
...
H: exec nil s s2
=========
   s=s2
```

```
inversion H.
```

```
H: exec nil s s2
x: s=s2
===========
   s=s2
```

The compiler : from expr to code

```
Fixpoint comp (e : expr) :=
match e with
  Val i => (PUSH i)::nil
| Add e1 e2 => comp e1 ++ comp e2 ++ (ADD::nil)
end.
```

```
Eval compute in comp (Add (Val 5) (Add (Val 4) (Val 3))).
Push 5 :: Push 4 :: Push 3 :: ADD :: ADD :: nil
     : list Op
```

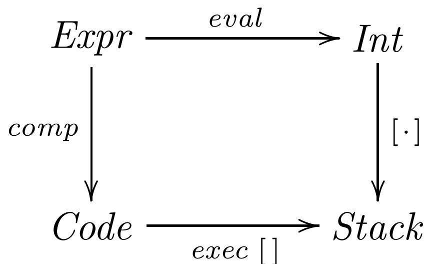
Compiler verification

The problem

Verify that a compiler is semantics-preserving:

the generated code behaves as prescribed by the semantics of the source program.

Prove the diagram (from Hutton & Wright) commutes !



Generalisation of the theorem:

```
Theorem gen_comp_correctness: forall e ops s f,
exec ((comp e)++ops) s f -> exec ops ((eval e)::s) f.
```

Proof technics here deserves to be explained. It will be applied again in the extension.

☺ Proving helps finding errors ! (I first swapped `i` and `j` in `exadd`, I discovered here).

Proof by induction on e.

Look at the recursive case (i.e. `e = add e1 e2`).

```
IHe1 : forall (ops : list Op) (s f : stack),
    exec (comp e1 ++ ops) s f -> exec ops (eval e1 :: s) f
IHe2 : forall (ops : list Op) (s f : stack),
    exec (comp e2 ++ ops) s f -> exec ops (eval e2 :: s) f
H : exec ((comp e1 ++ comp e2 ++ ADD :: nil) ++ ops) s f
============================
   exec ops (eval e1 + eval e2 :: s) f
```

The idea is to make appear

```
Hyp1 : exec (ADD :: ops) (eval e2 :: eval e1 :: s) f
```

in the context (`assert ... as Hyp1`).

Then by inversion of `Hyp1`, the conclusion will follow.

Proof of `Hyp1`:

```
IHe1 : forall (ops : list Op) (s f : stack),
    exec (comp e1 ++ ops) s f -> exec ops (eval e1 :: s) f
IHe2 : forall (ops : list Op) (s f : stack),
    exec (comp e2 ++ ops) s f -> exec ops (eval e2 :: s) f
H : exec ((comp e1 ++ comp e2 ++ ADD :: nil) ++ ops) s f
============================
   exec (ADD :: ops) (eval e2 :: eval e1 :: s) f
```

`Hyp1` derives from `IHe2` (`apply IHe2`)
and we have to prove

```
 exec (comp e2 ++ ADD ::  ops) (eval e1 ::  s) f
```

that we can solve by applying IHe1.
It requires the proof of

```
 exec (comp e1 ++ comp e2 ++ ADD ::  ops) s f.
```

It is nearly `H` (a small rewriting step in `H` is needed).

Adding exceptions

In `expr`: we add an expression that throws an exception and a catch expression that contains an handler.
e.g. `4 + (Catch Throw 5)`.

```
Inductive  expr : Set :=
   Val : nat -> expr
| Add : expr -> expr -> expr.
| Throw : expr
| Catch : expr -> expr -> expr.

Eval compute in Add (Val 4) (Catch Throw (Val 5)).
```

Adding exceptions

Denotational semantics

⟶ The value of a program is either an exceptional value or a natural number : `option nat`.

- `None` will be used to encode the exceptional value,
- `Some i` will be used to encode the normal value `i`.

⟶ `eval` has the type `expr -> option nat`.

```
Fixpoint  eval (e : expr) : option nat :=
match e with
  Val i => Some i    normal value
| Add x y =>
     match eval x with         propagation of the exception
       None => None
     | Some i => match eval y with
                              None => None
                            | Some j => Some (i+j)
                            end
     end
| Throw => None            exceptional value
| Catch x h => match eval x with
                 None => eval h          go on with h if x
               | Some i => Some i        throws an exception
               end
end.
```

```
Eval compute (eval (Add (Val 1) (Catch Throw (Val 4))).
Some 5: option nat

Eval compute (eval (Add (Val 1) (Add (Val 4) Throw))).
None : option nat.
```

Virtual machine code

Syntax

Adding a notion of address (represented as natural numbers)

```
Definition add := nat.
```

Adding of new primitives, `THROW`, `MARK a`, `UNMARK`, `LABEL a`, `JUMP a` where a is an adress.

```
Inductive  instr : Set :=
  PUSH : nat -> instr | ADD: instr |
  THROW : instr | MARK : add -> instr |  UNMARK : instr |
  LABEL : add -> instr | JUMP :  add -> instr.
```

```
Definition  code :=  list instr.
```

The stack will contain numbers and addresses of handlers.

```
Inductive item : Set  := VAL : nat -> item |
                         HAN : add -> item.
```

```
Definition stack := list item.
```

VM code

Semantics

Informally,

- `THROW` : stop the current computation, unwinds the stack until finding an handler and then executes it
- `MARK l`: push the address `l` on top of the stack
- `UNMARK`: remove the handler
- `Jump l`: transfer control to the code labelled by the address `l`

Transfer of control : `jump l cs` returns the part of `cs` that starts with `Label l`.

```
Fixpoint jump (l:add) (c : code):=
match c with
  nil => nil
| (Label l1)::cs =>
    if eq_nat_dec l l1 then cs else jump l cs
| c::cs => jump l cs
end.
```

```
Mutual Inductive exec : code -> stack -> stack -> Prop :=
 exnil : forall s, exec nil s s
|expush :  forall cs i s f, exec cs ((VAL i)::s) f ->
                                 exec ((PUSH i)::cs) s f
|exadd : forall cs i j s f, exec cs ((VAL(i+j))::s) f ->
    exec (ADD::cs) ((VAL j)::(VAL i):: s) f
|exthrow: forall cs s f, unwind cs s f -> exec (THROW::cs) s f
|exmark : forall l cs s f, exec cs ((HAN l)::s) f ->
                               exec ((MARK l)::cs) s f
|exunmark: forall cs v l s f, exec cs (v::s) f ->
    exec (UNMARK::cs) (v:: (HAN l) :: s) f
|exlabel: forall l cs s f, exec cs s f ->
                               exec ((LABEL l)::cs) s f
|exjump: forall l cs s f,  exec (jump l cs) s f ->
                               exec ((JUMP l)::cs) s f

with unwind : code -> stack -> stack -> Prop :=
 unnil: forall cs, unwind cs nil nil
|unval: forall cs i s f,  unwind cs s f ->
                              unwind cs ((VAL i)::s) f
|unhan: forall cs l s f,  exec (jump l cs) s f ->
                              unwind cs ((HAN l)::s) f.
```

Mutual recursive: not so easy to manipulate $\Longrightarrow$ turn into a unique one exec2 with a boolean to distinghuish between exec and exec.

```
Inductive exec2 : bool -> code -> stack -> stack -> Prop :=
  extnil : forall s, exec2 true nil s s
| extpush :  forall cs i s f, exec2 true cs ((VAL i)::s) f ->
                                exec2 true ((PUSH i)::cs) s f
| extadd :  ...
| extthrow: forall cs s f, exec2 false cs s f ->
                             exec2 true (THROW::cs) s f
| extmark : ...
| extunmark: ...
| extlabel: ...
| extjump: ...
| exfnil: forall cs, exec2 false cs nil nil
| exfval: forall cs i s f,  exec2 false cs s f ->
                              exec2 false cs ((VAL i)::s) f
| exfhan: forall cs l s f,  exec2 true (jump l cs) s f ->
                              exec2 false cs ((HAN l)::s) f.

Definition exec := exec2 true.
Definition unwind := exec2 false.
```

Compiler

We'll need to create fresh adresses $\Rightarrow$ counter

```
compile : add -> expr -> code*add

Fixpoint compile (l: add) (e: expr) :=
match e with
  Val i =>  ((Push i)::nil, l)
| Add x y => let (xs,m):= compile l x in
             let (ys,n):= compile m y in
             (xs ++ ys ++ (ADD::nil), n)
| Throw => (THROW::nil,l)
| (Catch x h) =>
     let (xs,m) := compile (l+2) x in
     let (hs,n) := compile m h in
     (((MARK l)::xs) ++
        (UNMARK :: JUMP (l+1) :: LABEL l :: nil) ++
        hs ++ (LABEL (l+1)::nil), n)
end.
```

An example

The expression $3 + (\text{catch } (1 + \text{Throw}) \; 2)$ is encoded as

```
Add (Val 3) (Catch (Add (Val 1) Throw) (Val 2))
```

It is compiled into

```
(*Eval compute in
(comp 0 (Add (Val 3) (Catch (Add (Val 1) Throw) (Val 2))))

PUSH 3 :: MARK 0 :: PUSH 1 :: THROW :: ADD :: UNMARK
 :: JUMP 1 :: LABEL 0 :: PUSH 2 :: LABEL 1 :: ADD :: nil
```

Execution on the stack machine

stack	*code*
3	$MARK\ 0; \ldots.$
$H\ 0 \vert 3$	$PUSH\ 1; \ldots.$
$1 \vert H\ 0 \vert 3$	$THROW; \ldots$
3	$PUSH\ 2; \ldots$
$2 \vert 3$	$LABEL\ 1; ADD$
$2 \vert 3$	ADD
5	nil

Compiler

Some facts about fresh addresses

- The "counter" for adresses in `compile` increases

  ```
  Lemma lem6 : forall e l, (l <=  snd(compile l e)).
  ```

- An address is fresh wrt a stack `s` if it is greater that all the adresses in `s`.

  ```
  Inductive isFresh : add -> stack -> Prop :=
    freshnil : forall l, isFresh l nil
  | freshVal : forall l s i, isFresh l s ->
              isFresh l ((VAL i)::s)
  | freshHan :  forall l l' s, (l' < l) ->
              isFresh l s -> isFresh l ((HAN l')::s).
  ```

- Some lemmas about freshness

```
Lemma lem5: forall l s m,  isFresh l s ->  (l <= m) ->
isFresh m s.

Lemma corol5_6: forall l s e, isFresh l s ->
isFresh (snd(compile l e)) s.

Lemma lem3_3bis : forall s l  f cs,  isFresh l s ->
(unwind cs s f <->  unwind ((Label l) :: cs) s f).
```

Compiler Verification

The correctness theorem

- We prove the general statement introducing addditional code.
- Remember the simple case:

```
Theorem gen_comp_correctness: forall e ops s f,
exec ((comp e)++ops) s f -> exec ops ((eval e)::s) f.
```

- Here we have to distinghuish 2 cases:
  - `e evaluates into a normal value` : then we push its value on top of the stack (idem as previously) before executing the additional code
  - `e evaluates into an exceptional value`: then we unwind the stack and transfer control to the next handler.
- We describe these both behaviors with an inductive predicate:

```
Inductive conv: stack -> option nat -> code -> stack -> Prop
  convSome : forall ops v s f, exec ops ((VAL v)::s) f ->
             conv s (Some v) ops f
| convNone : forall ops s f, unwind ops s f ->
             conv s None ops f.
```

We obtain;

```
Lemma compiler_correctness : forall e l ops s f,
isFresh l s ->
exec ((comp l e)++ops) s f ->
conv s (eval e) ops f.
```

Proof by induction on e using the following intermediate lemmas.
Add and Catch: tedious (all the cases of evaluation for the 2 arguments).

```
Lemma lem4 : forall e l s f cs,
isFresh l s ->  unwind ((comp l e)++cs) s f ->  unwind cs s f.

Lemma lem7 : forall e l m cs, (l < m) ->
    jump l ((comp m e) ++ cs) = jump l cs.
```

$\sim$ 600 lines (programs, spec. and proofs) (It could be made a little bit shorter by introducing Ltac tactics, I'll do it next week ... Look at the TOOLS Web site)

QED.