Michelson in Coq

Raphaël Cauderlier

January 8, 2018

Presentation

Raphaël Cauderlier

- PhD defended in 2016
  - directed by Catherine Dubois
  - at Deducteam (Inria, LSV, CNAM)
- Postdoc at IRIF (Paris 7) with Mihaela Sighireanu
- Joining Nomadic on February 25

Outline

What?

Define in Coq the following parts of Michelson:

- syntax
- typing
- semantics (evaluator)

Why?

- Fun
- Many applications:
  - Check the Michelson specification
  - Prove properties of Michelson programs
  - Prove correctness of compilers from/to Michelson
  - Extract some correct-by-construction tools

Related work

`https://github.com/tezos/tezoscoq`

- + Proof of the multisig contract
- - Untyped instructions
- - Out of date (last commit one year ago)

Outline

The Coq interactive theorem prover

- Developped for more than 30 years
- Non-trivial mathematical theorems: 4-color, odd-order
- CompCert: certified C compiler

Dependent Types

```
Parameter A : Type.
```

Dependent Types

```
Parameter A : Type.

Parameter a : A.
```

Dependent Types

```
Parameter A : Type.

Parameter a : A.

Parameter B : A -> Prop.
```

Dependent Types

```
Parameter A : Type.
Parameter a : A.
Parameter B : A -> Prop.
Parameter f : forall x : A, B x.
```

Dependent Types

```
Parameter A : Type.

Parameter a : A.

Parameter B : A -> Prop.

Parameter f : forall x : A, B x.

Check f a.  (* Answer: f a : B a *)
```

Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

Inference of first argument

```
Lemma identity_2 : identity _ 2 = 2.
Proof. reflexivity. Qed.
```

Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

Inference of first argument

```
Lemma identity_2 : identity _ 2 = 2.
Proof. reflexivity. Qed.
```

Implicit argument

```
Definition id {A : Type} (a : A) := a.

Lemma id_2 : id 2 = 2.
Proof. reflexivity. Qed.
```

Inductive Types

Generalisation of ADT to dependent types:

```
Inductive vector (A : Type) : nat -> Type :=
  | Nil : vector A 0
  | Cons n : A -> vector A n -> vector A (1 + n).
```

Outline

Outline

Code

https://framagit.org/rafoo/michelson-coq

Types

```
Inductive comparable_type : Set := nat | int ...

Inductive type : Set :=
| Comparable_type : comparable_type -> type
| unit : type
| set : comparable_type -> type
...

Coercion Comparable_type : comparable_type >-> type.
```

Types

```
Definition comparable_data (a : comparable_type)
   : Set :=
  match a with
  | nat => N
  | int => Z
  ...
  end.

Fixpoint data (a : type) {struct a} : Set :=
  match a with
  | Comparable_type b => comparable_data b
  | unit => Datatypes.unit
  ...
  end.
```

Syntax

```
Inductive instruction : list type -> list type -> Set :=
| FAILWITH {A B a} : data a -> instruction A B
| SEQ {A B C} : instruction A B -> instruction B C ->
                instruction A C
| IF {A B} : instruction A B -> instruction A B ->
             instruction (bool ::: A) B
| LOOP {A} : instruction A (bool ::: A) ->
             instruction (bool ::: A) A
...
```

Syntax

```
Inductive instruction : list type -> list type -> Set :=
| FAILWITH {A B a} : data a -> instruction A B
| SEQ {A B C} : instruction A B -> instruction B C ->
                instruction A C
| IF {A B} : instruction A B -> instruction A B ->
             instruction (bool ::: A) B
| LOOP {A} : instruction A (bool ::: A) ->
             instruction (bool ::: A) A
...
```

Syntax and typing simultaneously

Semantics

```
Fixpoint eval {A : list type} {B : list type}
      (i : instruction A B) : stack A -> stack B :=
    match i in instruction A B
      return stack A -> stack B with
    | FAILWITH x =>
       ...
    | SEQ i1 i2 =>
       fun SA => eval i2 (eval i1 SA)
    | IF bt bf =>
       fun SbA => let (b, SA) := SbA in
         if b then eval bt SA else eval bf SA
    | LOOP body =>
       fun SbA => let (b, SA) := SbA in
         if b then eval (SEQ body (LOOP body)) SA
         else SA
    ...
```

Semantics

```
Fixpoint eval {A : list type} {B : list type}
      (i : instruction A B) : stack A -> M (stack B) :=
    match i in instruction A B
      return stack A -> M (stack B) with
    | FAILWITH x =>
       fun SA => Failed _ (Assertion_Failure _ x)
    | SEQ i1 i2 =>
       fun SA => bind (eval i2) (eval i1 SA)
    | IF bt bf =>
       fun SbA => let (b, SA) := SbA in
         if b then eval bt SA else eval bf SA
    | LOOP body =>
       fun SbA => let (b, SA) := SbA in
         if b then eval (SEQ body (LOOP body)) SA
         else Return _ SA
    ...
```

Semantics

```
Fixpoint eval {A : list type} {B : list type}
      (i : instruction A B) (fuel : nat)
      {struct fuel} : stack A -> M (stack B) :=
  match fuel with
  | O => fun SA => Failed _ Out_of_fuel
  | S n =>
    match i in instruction A B
      return stack A -> M (stack B) with
    | FAILWITH x =>
       fun _ => Failed _ (Assertion_Failure _ x)
    | SEQ i1 i2 =>
       fun SA => bind (eval i2 n) (eval i1 n SA)
    | IF bt bf =>
       ...
    | LOOP body =>
       ...
```

Fuel vs Gas

- Fuel: Coq trick to turn a non-terminating function into a terminating one
  - Fuel (SEQ A B) = max(Fuel A, Fuel B) + O(1)
- Gas: measures the complexity of the program
  - Gas (SEQ A B) = Gas A + Gas B + O(1)

Discussion

- Separating syntax and semantics
  - Close to the Michelson compiler written in OCaml
  - Does not scale very well
- Modular presentation (instruction by instruction)
  - Close to the specification
  - Useful to handle overloading

Outline

Summary

- Michelson is a simple language
  No major difficulty to formalize it in Coq
- A few small mistakes detected in the specification

Evolution

- Extract the evaluator
- Share the formalisation of the syntax with the current compiler
- Extract the documentation of Michelson from its formalisation

Thank you!

Questions?

Semantics of the data types

Michelson	Coq	Michelson	Coq
`int`	`Z`	`pair a b`	`a * b`
`nat`	`N`	`option a`	`option a`
`string`	`string`	`or a b`	`sum a b`
`bytes`	`string`	`list a`	`list a`
`timestamp`	`Z`	`set a`	`set a (lt a)`
`mutez`	`int63`	`map a b`	`map a b (lt a)`
`bool`	`bool`	`bigmap a b`	idem
`unit`	`unit`	`lambda a b`	`a -> M b`
		anything else	axiomatized

with

```
Definition int63 :=
  {t : int64.int64 | int64.sign t = false}
Definition set a lt :=
  {l : list A | Sorted.StronglySorted lt l}
Definition map a b lt :=
  set (a * b) (fun x y => lt (fst x) (fst y))
```

Overloading

Almost fully supported using canonical structures.

```
Module neg.
  Record class (a : comparable_type) :=
    Class { neg : comparable_data a -> M Z }.
  Structure type (a : comparable_type) :=
    Pack { class_of : class a }.
  Definition op (a : comparable_type) {e : type a}
    : comparable_data a -> M Z := neg _ (class_of a e).
End neg.

Canonical Structure neg_nat : neg.type nat :=
  neg.Pack nat (neg.Class nat
    (fun x => Return _ (- Z.of_N x)%Z)).
Canonical Structure neg_int : neg.type int :=
  neg.Pack int (neg.Class int
    (fun x => Return _ (- x)%Z)).
```