A Verified Compiler for an Impure Functional Language

水野 雅之

2015 年 12 月 24 日

Motivation

Planning to verify MinCaml for graduate thesis
There are several previous reserches (such as CompCert) but...

MinCaml features

- First class functions
- Impure operations
- Foreign function interface

Thus tried to survey closely related paper
"A Verified Compiler for an Impure Functional Language"

A Verified Compiler for an Impure Functional Language?

Adam Chlipala, POPL10

verifies impure functional language with Coq

$$
\begin{array}{rcl}
e & ::= & c \mid e = e \mid x \mid e\ e \mid \texttt{fix}\ f(x).\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid () \mid \langle e, e \rangle \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \texttt{inl}(e) \mid \texttt{inr}(e) \\
& & \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \Rightarrow e \mid \texttt{inr}(x) \Rightarrow e \\
& & \mid \texttt{ref}(e) \mid\ !e \mid e := e \mid \texttt{raise}(e) \mid e\ \texttt{handle}\ x \Rightarrow e
\end{array}
$$

$\rightarrow$ close to MinCaml's source language

Outline

Outline

Introduction

A lot of sucks around mechanized proof

- wrong abstractions (e.g. nested variable binders)
- copious lemma and case analysis
- Once add a new constructor, Modify proof anywhere

Proposing "engineering" approach to reduce development costs

- *Parametric Higher-Order Abstract Syntax*
- new semantic approach
- sophisticated proof automation

Source Language

untyped, subset of ML

$$
\begin{aligned}
e \;::=\;& c \mid e = e \mid x \mid e\ e \mid \texttt{fix}\ f(x).\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& \mid () \mid \langle e, e \rangle \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \texttt{inl}(e) \mid \texttt{inr}(e) \\
& \mid \texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \Rightarrow e \mid \texttt{inr}(x) \Rightarrow e \\
& \mid \texttt{ref}(e) \mid\ !e \mid e := e \mid \texttt{raise}(e) \mid e\ \texttt{handle}\ x \Rightarrow e
\end{aligned}
$$

no variable-arity features (e.g. sum, product)

no compound pattern matching

Target Assembly Language Syntax

idealized assembly language

$$
\begin{array}{rcl}
r & ::= & r_0 \mid \ldots \mid r_{N-1} \\
n & \in & \mathbb{N} \\
L & ::= & r \mid [r+n] \mid [n] \\
R & ::= & n \mid r \mid [r+n] \mid [n] \\
I & ::= & L ::= R \mid r \stackrel{+}{=} n \mid L := R \stackrel{?}{=} R \mid \texttt{jnz}\ R, n \\
J & ::= & \texttt{halt}\ R \mid \texttt{fail}\ R \mid \texttt{jmp}\ R \\
B & ::= & (I^*, J) \\
P & ::= & (B^*, B)
\end{array}
$$

finite registers and no interface for memory managements

Source Language Semantics

big step operational semantics $(h_1, e) \Downarrow (h_2, r)$

$$\frac{}{(h, \texttt{fix}\ f(x).\ e) \Downarrow (h, \texttt{Ans}(\texttt{fix}\ f(x).\ e))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \texttt{Ans}(\texttt{fix}\ f(x).\ e)) \qquad (h_2, e_2) \Downarrow (h_3, \texttt{Ans}(e')) \qquad (h_3, e[f \mapsto \texttt{fix}\ f(x).\ e][x \mapsto e']) \Downarrow (h_4, r)}{(h_1, e_1\ e_2) \Downarrow (h_4, r)}$$

$$\frac{(h_1, e) \Downarrow (h_2, \texttt{Ex}(v))}{(h_1, e_1\ e_2) \Downarrow (h_3, \texttt{Ex}(v))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \texttt{Ans}(\texttt{fix}\ f(x).\ e)) \qquad (h_2, e_2) \Downarrow (h_3, \texttt{Ex}(v))}{(h_1, e_1\ e_2) \Downarrow (h_3, \texttt{Ex}(v))}$$

$$\frac{(h_1, e) \Downarrow (h_2, \texttt{Ans}(v))}{(h_1, \texttt{ref}(e)) \Downarrow (v :: h_2, \texttt{Ans}(\texttt{ref}(|h_2|)))}$$

$$\frac{(h_1, e) \Downarrow (h_2, \texttt{Ans}(\texttt{ref}(n))) \qquad h_2.n = v}{(h_1, !e) \Downarrow (h_2, \texttt{Ans}(v))}$$

Source Language Semantics

$$\frac{(h_1, e) \Downarrow (h_2, \texttt{Ans}(v))}{(h_1, \texttt{raise}(e)) \Downarrow (h_2, \texttt{Ex}(v))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \texttt{Ex}(v)) \qquad (h_2, e_2[x \mapsto v]) \Downarrow (h_3, r)}{(h_1, e_1 \ \texttt{handle}\ x \Rightarrow e_2) \Downarrow (h_3, r)}$$

Theorems ignore non-termination.

Outline

Treating nested variable binders concretely is confused

- capture (e.g. $[x \mapsto y](\lambda y.\ x)$)
- shadowing (e.g. $[x \mapsto y](\lambda x.\ x)$)

```
Inductive exp : Type :=
  | Var : string -> exp
  | App : exp -> exp -> exp
  | Abs : string -> exp -> exp.

Check (Abs ""x"" (Abs ""y"" (Var ""x""))).
```

Embedded binders into meta language

```
Inductive exp : Type :=
  | App : exp -> exp -> exp
  | Abs : (exp -> exp) -> exp.

Check (Abs (fun x => Abs (fun y => x))).
```

But Coq does not accept it because of non-termination

Does this OCaml program terminate?

```
type exp =
  | App of exp * exp
  | Abs of (exp -> exp)

let delta (Abs m as n) = m n;;
delta (Abs delta);;
```

HOAS sucks

Hard to deconstruct

How do we pretty-print "Abs (fun x $\to$ x)"?

```
let rec to_string : exp -> string = function
  | App (m, n) -> to_string m ^ "␣" ^ to_string n
  | Abs f -> (* !!!! *)
```

Parametric Higher-Order Abstract Syntax

Represent binders as function over variable

Guarantee well-formedness using parametricity

```
Inductive exp var : Type :=
  | Var : var -> exp var
  | App : exp var -> exp var -> exp var
  | Abs : (var -> exp var) -> exp var.

Definition Exp := forall var : Type, exp var.
Check (fun var => Abs var (fun x => Abs var (fun y => Var var x))).
```

Coq accept it

happiness of PHOAS

easier to deconstruct

```
let rec to_string : string exp -> string = function
  | Var x -> x
  | App (m, n) -> to_string m ^ "␣" ^ to_string n
  | Abs f -> let x = gensym () in "(fun␣" ^ x ^ "␣->␣" ^ f x ^ ")"
```

substitution is easily implementable

```
Fixpoint flatten var (e : exp (exp var)) : exp var :=
  match e with
  | Var x => x
  | App m n => App _ (flatten _ m) (flatten _ n)
  | Abs f => Abs _ (fun x => flatten _ (f (Var x)))
  end.
Definition Subst (E : forall var, var -> exp var) (E' : Exp) : Exp
  := fun var => flatten (E _ E').
```

PHOAS specific features

Correctness proofs usually parametricity
Axiomatize equivalence and Formalize parametricity

$$\frac{(x_1, x_2) \in \Gamma}{\Gamma \vdash \#x_1 \sim \#x_2}$$

$$\frac{\Gamma \vdash e_1 \sim e_1' \qquad \Gamma \vdash e_2 \sim e_2'}{\Gamma \vdash e_1\ e_2 \sim e_1'\ e_2'}$$

$$\frac{\forall x_1 x_2.\ \Gamma, (x_1, x_2) \vdash f_1(x_1) \sim f_2(x_2)}{\Gamma \vdash \lambda f_1 \sim \lambda f_2}$$

Definition 1 (well-formedness)

E is well-formed if, for any $\mathtt{var}_1$ and $\mathtt{var}_2$, we have
$\cdot \vdash E(\mathtt{var}_1) \sim E(\mathtt{var}_2)$

Outline

Define semantics over instantiated term (i.e. exp val)

Coq rejects naive definition of value

```
Inductive val : Type :=
  | VAbs : (val -> exp val) -> val.
```

Solution: Represent value as pointer to closure

```
Definition val := nat.
Definition closure := val -> exp val.
Definition heap := list closure.
```

Substitution-Free Operational Semantics

This technique can be generalized to the full source language

```
Inductive val : Type :=
  | VFunc : label -> val
  | VUnit : val
  | VPair : val -> val -> val
  | VInl : val -> val
  | VInr : val -> val
  | VRef : label -> val.
```

Surprisingly, the change reduces hassle in mechanized proofs

Outline

PHOASification

To let final theorem independent of PHOAS, beginning with a de Bruijn index

Translate de Bruijn index style programs into PHOAS style

$$
\begin{aligned}
\lfloor \#x \rfloor \sigma &= \#(\sigma.x) \\
\lfloor e_1\ e_2 \rfloor \sigma &= \lfloor e_1 \rfloor \sigma\ \lfloor e_2 \rfloor \sigma \\
\lfloor \texttt{fix}\ f(x).\ e_1 \rfloor \sigma &= \texttt{fix}(\hat{\lambda} f.\ \hat{\lambda} x.\ \lfloor e_1 \rfloor (x :: f :: \sigma))
\end{aligned}
$$

Formalize correspondence and prove monotonicity

Lemma 1

For any e and σ with compatible type indices, if e contains no uses of \$, then $\cdot, \sigma \vdash e \simeq \lfloor e \rfloor \sigma$

Lemma 2

If $(h_1, e) \Downarrow (h_2, r)$ at source level, $H, \sigma \vdash e \simeq e'$ and $H \vdash h_1 \simeq h_1'$, then there exist H', h_2' and r' such that $(H, h_1', e') \Downarrow (H', h_2', r')$, $H' \vdash r \simeq r'$ and $H' \vdash h_2 \simeq h_2'$

CPS Translation

convert syntax trees to CPS syntax tree

$$
\begin{array}{rcl}
p & ::= & c \mid x = x \mid \texttt{fix}\ f(x).\ e \mid () \mid \langle x, x \rangle \mid \texttt{inl}(x) \mid \texttt{inr}(x) \\
& & \mid \texttt{ref}(x) \mid\ !x \mid x := x \\
e & ::= & \texttt{halt}(x) \mid \texttt{fail}(x) \mid x\ x \mid \texttt{let}\ x = p\ \texttt{in}\ e \\
& & \mid \texttt{case}\ x\ \texttt{of}\ \texttt{inl}(x) \Rightarrow e \mid \texttt{inr}(x) \Rightarrow e
\end{array}
$$

$$
\begin{array}{rcl}
\lfloor \#x \rfloor k_S k_E & = & k_S(x) \\
\lfloor \texttt{raise}(e) \rfloor k_S k_E & = & \lfloor e \rfloor (\overset{\wedge}{\lambda}\, x.\ k_E(x)) k_E \\
\lfloor \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rfloor k_S k_E & = & \lfloor e_1 \rfloor (\overset{\wedge}{\lambda}\, x. \lfloor e_2 \rfloor k_S k_E) k_E
\end{array}
$$