Certifying a compiler for an imperative language using Coq

Yves Bertot

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93
06902 Sophia Antipolis CEDEX

November 1998

Motivation

– a complement to a study on programming language semantics (some of it done by Ranan Fraer)

  * certified rewriting transformations,
  * certified elementary partial evaluation,
  * type conservation,
  * coherence between natural semantics (large step) and structural operational semantics (small step).

– a less abstract target language (when compared to Mini-ML & CAM),

  * type confusion,
  * structure removal and `goto` instructions

2 Language descriptions

The source language

– 2 datatypes,

– simple unary and binary operators (all strict),

– 4 instructions: assignment, conditional, while loop, sequence.

The target language

- A machine with one register, infinite stack and memory.
- One data type : unbounded natural numbers.
- no program counter: use of lists of instructions.
- stack and register management: `push`, `pop`, `load`, `store`,
- labels, conditional and unconditional branching: `label`, `goto`, `branch_if_0`,
- arithmetical and logical instructions: `and`, `not`, `add`, `sub`, `gt`, `eq`,
- no `or` instruction, although `or` is present in the source language.

4 Language descriptions

source language semantic description

– Natural semantics descriptions of the source language in the Centaur system.

$$\frac{\rho \vdash E \rightarrow \texttt{true} \qquad \rho \vdash I_1 \Rightarrow \rho'}{\rho \vdash \text{if } E \text{ then } I_1 \text{ else } I_2 \Rightarrow \rho'}$$

– Automatic translation to Coq, thanks to D. Kaplan, resulting in inductive properties.

– Automatic translation to Prolog, thanks to T. Despeyroux, resuting in an interactive environment to interprete (simulate) programs.

5 Language descriptions

target language semantic description

- elementary operation described using total functions,
- partial functions represented by a sum result type,
- complete program execution described by an inductive property,
- arithmetic and boolean operations taken directly from Coq operations on natural numbers.

semantic details (source language)

Inductive $exec$:

$DECLS \Rightarrow INST \Rightarrow DECLS \Rightarrow Prop$

$:=$

...

exec_if_true:

$\forall i_1, i_2 : INST.\ \forall e : EXP.\ \forall d, d' : DECLS.$
$eval(d, e, true) \Rightarrow exec(d, i_1, d') \Rightarrow$
$\quad exec(d, if(e, i_1, i_2), d')$

...

exec_while_true:

$\forall i : INST.\ \forall E : EXP.\ \forall d, d', d'' : DECLS.$
$eval(d, e, true) \Rightarrow exec(d, i, d') \Rightarrow$
$exec(d', while(e, i), d'') \Rightarrow$
$\quad exec(d, while(e, i), d'')$

semantic details (target language)

$exec\_state$ is a triple type: memory, stack and register.

Inductive $execute$ $[pg : list(Assembly)]$:
$$exec\_state \rightarrow list(Assembly) \rightarrow$$
$$exec\_state \rightarrow Prop$$
$:=$

finish :
$$\forall st : exec\_state.\ execute(pg, st, [], st)$$

one_step :
$$\forall p_1, p_2 : list(Assembly).\ \forall i : Assembly \ldots$$
$$new\_pc(pg, p_1, r_1, i) = Some(p_2) \Rightarrow$$
$$new\_stack(r_1, s_1, i) = Some(s_2) \Rightarrow$$
$$new\_register(r_1, m1, s_1, i) = Some(r_2) \Rightarrow$$
$$execute(pg,$$
$$\langle new\_mem(r_1, m_1, i), s_2, r_2\rangle, p_2, st) \Rightarrow$$
$$execute(pg, \langle m_1, s_1, r_1\rangle, i \cdot p_1, st).$$

8 Language descriptions

Compiler code

Definition $compile\_bin :=$
$[p_1, p_2, p_3 : list(Assembly)]\ p_1 \cdot push \cdot p_2 \cdot p_3$.

Fixpoint $compile\_exp\ [e : EXP]$: list(Assembly) :=
Cases e of
$ident(n) \Rightarrow load(n)$
$v \Rightarrow immediate(compile\_val(v))$
(* v is a value *)
$\ldots$
$and(i_1, i_2) \Rightarrow$
$compile\_bin(compile\_exp(i_1),$
$compile\_exp(i_2), and)$
$plus(i_1, i_2) \Rightarrow$
$compile\_bin(compile\_exp(i_1),$
$compile\_exp(i_2), plus)$

9 Language descriptions

Compiler code *(continued)*

Fixpoint compile_inst $[i : INST]$
$: nat \rightarrow (list(Assembly) * nat) :=$
Cases i of
. . .
$while(e, i') \Rightarrow$
let $(p, n') = compile\_inst(i', n+2)$ in
$(label(n) \cdot compile\_exp(e) \cdot$
$branch\_if\_O(n+1) \cdot p \cdot goto(n) \cdot$
$label(n+1), n')$
. . .

10 Proof organisation

Preliminary remarks

– For each language, reason on proof derivations for the main inductive predicate to reason on execution,

– For each language, the proof derivation structure is influenced by the language structure (thanks to natural semantics).

– For expression compiling, a systematic treatement of binary operators.

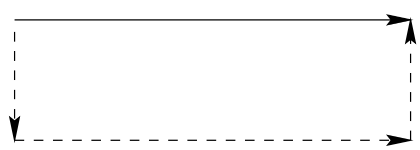
– Structure is lost in the compiling process.

– A careful treatement of label indices.

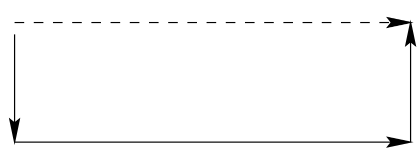
11 Proof organisation

$\boxed{\textbf{Two proofs}}$

– Any program that terminates compiles and the compiled program does the same computation (easy).



– If a program compiles and the compiled program terminates, then the program's behavior can be simulated using the source-level semantics.

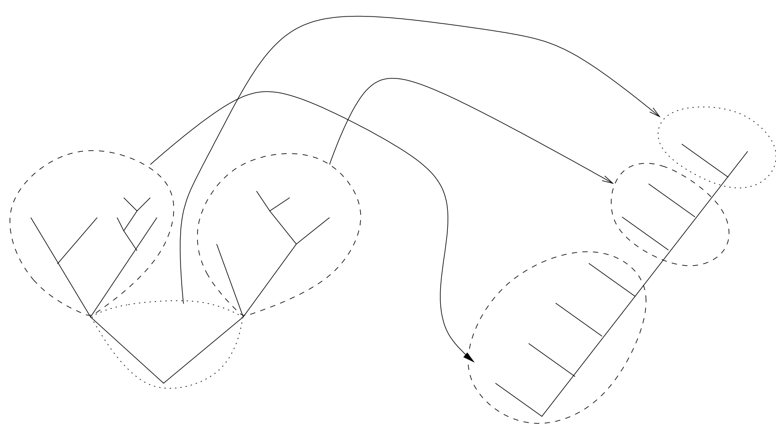


Properties proved only for well-typed programs.

Differences

Proofs are structured by the derivations that one can assume, from this derivation one tries to build the other derivation.

– In the first proof, the known derivation is structured.



– In the second proof, the known derivation is not structured.

Fragmentation theorems

– important properties for assembly programs:

  ∗ unique labels,
  ∗ self-contained fragments.

– **Theorem:** self-contained fragments can be studied in isolation.

$$\forall p_1, p_2, p_3, s, s''.$$
$$unique(p_1 \cdot p_2 \cdot p_3) \wedge self\_contained(p_2) \Rightarrow$$
$$(\exists s'.execute(p_2, s, p_2, s') \wedge execute(p_1 \cdot p_2 \cdot p_3, s', p_3, s''))$$
$$\Leftrightarrow$$
$$execute(p_1 \cdot p_2 \cdot p_3, s, p_2 \cdot p_3, s'')$$

Abstract setting for fragmentation

– over-simplified abstract machine: `state`, `Inst`, `Lab`.

– computation represented by abstract function:
`lab_eq_dec` : `Lab` $\rightarrow$ `Lab` $\rightarrow$ $\rightarrow$ $bool$
`new_state` : `Inst` $\rightarrow$ `state` $\rightarrow$ `state`
`test_branch` : `Inst` $\rightarrow$ `state` $\rightarrow$ `bool`

– abstract assembly with 3 instructions:
`plain` : `Inst` $\rightarrow$ `abs_assembly`,
`branch` : `Inst` $\rightarrow$ `abs_assembly`,
`label` : `Lab` $\rightarrow$ `abs_assembly`

– execution:
`machine` :`(list abs_assembly)` $\rightarrow$ `state` $\rightarrow$ `(list abs_assembly)`$\rightarrow$ `state` $\rightarrow$ `Prop`,

15 Fragmentation

$\boxed{\textbf{concrete to abstract maps}}$

- α : `Assembly` $\rightarrow$ `abs_assembly`
- γ : `abs_assembly` $\rightarrow$ `Assembly`
- injective translation:

$$\gamma \circ \alpha = Id$$

- execution modeling :
  `machine` $\circ\ \alpha \Leftrightarrow$ `execute`

A compiler proof of its own!

Simple Induction Principles

– Primitive induction theorem : induction hypothesis only for direct subterms or derivations.

```
Inductive nat :  Set := O : nat
          S : nat → nat.

nat_ind :  ∀ P: nat → Prop.
   P(O) ⇒ ( ∀ m:nat.  P(m) ⇒ P(S(m))) ⇒
   ∀ n:nat.  P(n)

Inductive ≤ :  nat → nat → Prop :=
     le_n :  ∀ n:nat.  n ≤ n
       le_S : ∀ n, m:nat.  n ≤ m ⇒
             n ≤ S(m).

le_ind :  ∀ nat → nat → Prop.
   (∀ n:nat.  P(n, n)) ⇒
   (∀ n, m:nat.  P(n, m) ⇒ P(n, S(m))) ⇒
   ∀ n, m:nat.  n ≤ m ⇒ P(n, m)
```

General Induction on Natural Numbers

- $\forall$ `P: nat` $\longrightarrow$ `Prop.`
  `(`$\forall$ `x:nat.`
  `(`$\forall$ `y:nat. y < x` $\Rightarrow$ `P(y))`$\Rightarrow$ `P(x))`$\Rightarrow$
  $\forall$ `z:nat. P(z)`

- Proof: use the simple induction principle with Q:
  `Q(n)` $\equiv$ `P(n)` $\wedge$ `R(n)`
  `R(n)` $\equiv$ `(`$\forall$ `y:nat. y < x` $\Rightarrow$ `P(y))`

- `Q(n)` $\Rightarrow$ `R(S(n))`,

- with hypothesis `R(n)` $\Rightarrow$ `P(S(n))`,

- altogether `Q(n)` $\Rightarrow$ `Q(S(n))`.

General Induction on Assembly Execution

- predecessors of execute derivations: $\texttt{execute}'$ is analogous to < for pairs (state $\times$ program)
  beware: predecessors of derivations are actually successors of execution...

- $$(\texttt{execute}'(pg, s, p_1, s', p_2) \wedge \texttt{execute}(pg, s', p_2, s'')) \Leftrightarrow \texttt{execute}(pg, s, p_1, s'')$$

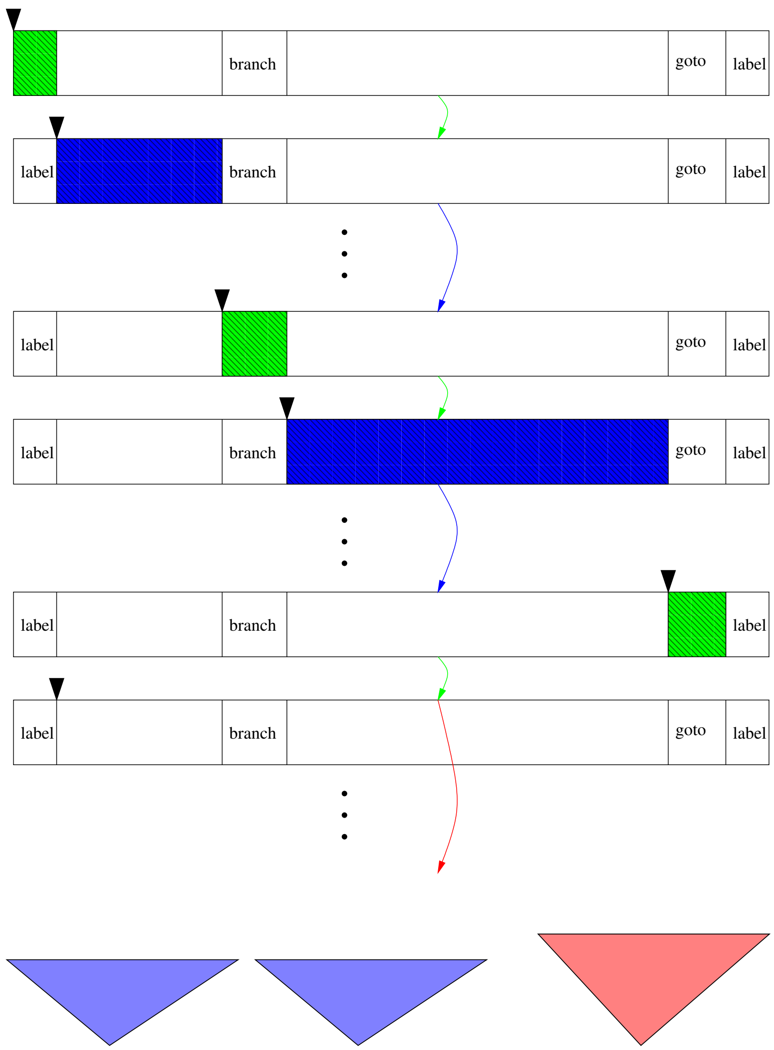
- $$\forall P. (\forall s, p, s'. (\forall s_1, p_1. \texttt{execute}'(pg, s, p, s_1, p_1) \wedge \texttt{execute}(pg, s_1, p_1, s') \Rightarrow P(s_1, p_1, s')) \Rightarrow P(s, p, s')) \rightarrow \forall s, p, s'. \texttt{execute}(pg, s, p, s') \Rightarrow P(s, p, s')$$

- proved using the same technique as for natural numbers.

Using the Generalized Induction Principle

- Only for `while` loops,
- Whole proof structured by a structural induction on programs,
- Structural induction inadequate for while loops (true case) only,

Using the Generalized Induction Principle
(continued)



$$\frac{\rho \vdash E \longrightarrow \text{true} \qquad \rho \vdash I \Longrightarrow \rho' \qquad \rho' \vdash \text{while } E \text{ do } I \Longrightarrow \rho''}{\rho \vdash \text{while } E \text{ do } I \Longrightarrow \rho''}$$

21 Conclusion

Facts

- extracted program 150 lines,
- proof and formalisation: 8000 lines, 200 lemmas,
- between 2 and 3 man-months,
- using CtCoq: proof-by-pointing, algebraic properties of list concatenation.

22 Conclusion

future work?

- partial automation,
  - use reflection techniques to handle uniqueness and self-containedness proofs.
  - mechanize the process of generating general induction.
- language extension,
  - extensions on datatype, arrays, type definitions.
  - procedures.
- adding more layers: use this language as target for another compiler.