SGoto in Coq

(Experience Report)

Reynald Affeldt

Research Center for Information Security (RCIS)
National Institute of Advanced Industrial Science and Technology (AIST)
http://www.rcis.aist.go.jp

First time online: Decembre 11, 2008; Last update: June 25, 2010

1 Motivation and Contribution

Motivation The main motivation for the formalization of SGoto [SU07] is the production of mechanically-checkable proofs of correctness for assembly programs. [SU07] actually provides two ways for producing such proofs. The first one is an original (compositional) Hoare logic that one can use directly to prove correctness. The second one is a compiler from structured programs (with while-loops) to programs with gotos that preserves the validity of Hoare triples (Theorem 17 in [SU07]). The latter is useful in situations where the traditional Hoare-logic proof already exists. This is often the case, since textbooks usually provide correctness arguments in terms of invariants for structured programs. [AM06] is a concrete example of such a situation. [AM06] proves in the Coq proof assistant the correctness of an implementation in the SmartMIPS instruction set of the Montgomery multiplication. The formal verification was done using Separation logic for a While-like language. The last step of the verification was to generate a ready-to-run assembly program by "compiling" while-loops into gotos. For this purpose, [AM06] provides such a "compiler" (this is rather a macro-expander) and proves in Coq its correctness, i.e., that it preserves the operational semantics. Yet, strictly speaking, that does not give a mechanically-checkable proof that the Separation-logic triple holds for the assembly program to be run.

Contribution [SU07] is a pencil-and-paper formalization for an archetypal assembly language. In this document, we not only formalize most of the pencil-and-paper proofs in [SU07] but we also instantiate them with a concrete instruction set (a subset of the SmartMIPS instruction set) and extend them with error-states (to model instructions that may trap). This enables the construction of mechanically-verifiable correctness proofs for realistic programs. The main differences between the assembly language we formalize and the archetypal assembly language of [SU07] are as follow:

- The store of variables consists of a finite number of finite-size registers. As a concrete consequence, we can only prove that the factorial program of [SU07] is correct modulo 2^{32} (see Section 6): this is an intended and desirable property.

- Besides a store, the state also comprises a mutable memory. Actually, the underlying logic is not just predicate logic but Separation logic [Rey02]. This enables the verification, for example, of programs for multi-precision arithmetic, as illustrated in Section 8.
- The operational semantics deals with error-states so as to model arithmetic overflows and unaligned memory accesses.

All these extensions are orthogonal to the formalization of [SU07], so that we are able to isolate cleanly the proofs of [SU07] from the details due to the concrete instruction set in use using Coq modules. This makes our formalization reusable.

Comparison with [SU07] Table 1 (page 3) makes it clear what is formalized w.r.t. [SU07]. In brief, what we do not do: we do not formalize Section 5 of [SU07] about decompilation (anyway, the topic is mentioned only briefly in [SU07]) and we formalize only the so-called "non-constructive proofs" of Theorems 17 and 18 (indeed, for these two theorems, the proofs come in two flavors).

As explained above, we instantiate the proofs of [SU07] with a concrete instruction set and with error-states. Error-states are responsible for longer proofs because they duplicate case-analyses. Besides length, proofs are essentially the same as [SU07]. The added value is the eradication of the inevitable typos and imprecisions of pencil-and-paper proofs, and also the fact that proofs in Coq can be replayed interactively.

Implementation Overview Table 2 (page 4) is a short overview of the implementation. For each file, we give the number of lines of Coq scripts (comments and blank lines removed). Compared with the 43 pages of [SU07] (accepted authors manuscript) and given the benefits of mechanization, these figures are reasonable. For reference, we also indicate the scripts for instantiation to SmartMIPS (taken from [AM06]).

The corresponding HTML documentation is available at `http://staff.aist.go.jp/reynald.affeldt/coqdev/cryptoasm.{filename_without_extension}.html`.

We use SSReflect [GM07] and, despite our awkward command of this Coq extension, we feel it improves readability and manageability.

The Rest of this Document The next sections are organized so as to match the organization of [SU07], with the part about the While language coming first (it was in appendix in [SU07]). The Coq code has been extracted directly from the Coq scripts using the coqdoc utility. Section 8 details an application to the proof of [AM06].

2 While: A Low-level Language

This section corresponds to Appendix A in [SU07].

Our formalization of [SU07] can be instantiated with any While-like language. In this section, we isolate more precisely what we expect from such a language.

2.1 Generic definition of then While Language and Hoare logic

`Section` *Lang.*

Reference in [SU07]	Status in “SGoto in Coq” (this document)
Section 2 Goto, a low-level language	
Figure 1	Done
Lemma 1	Done
Lemma 2	Particular cases only
Lemma 3	Done
Section 3 SGoto, a structured version	
Section 3.1 Syntax and natural semantics of SGoto	
Figure 2	Done
Lemmas 4–5	Done
Theorems 6–8	Done
Corollary 9	Done
Section 3.2 Hoare Logic of SGoto	
Figure 3	Done
Theorem 10	Done
Lemma 11	Done
Theorem 12	Done
Section 4 Compilation from While to SGoto	
Section 4.1 Compilation and preservation/reflection of evaluations	
Figure 5	Done
Lemmas 13–14	Done
Theorems 15–16	Done
Section 4.2 Preservation/reflection of derivable Hoare triples	
Theorems 17–18	Done (non-constructive proofs only)
Section 4.3 Example	
	Done
Section 5 Compilation from SGototo While	
	Not done
Appendix A The high-level language While	
	Done
Appendix B Full proofs of Theorems 6, 7, 15, 16, 17, 18	
	Done (except the constructive proofs of 17-18)

Table 1: Status of the Formalization

A state is a pair of a store and a mutable memory.

`Variable` *store* : `Set`.
`Variable` *heap* : `Type`.
`Let` *state* : `Type` := (*store* × *heap*)%type.

We are given one-step, non-branching instructions: `Variable` *cmd0* : `Set`.

One-step, non-branching instructions are given an appropriate operational semantics. We use an option type to model error-states.

`Variable` *exec0* : *option state* → *cmd0* → *option state* → `Prop`.
`Notation` "s '–' c '—->' t" := (*exec0 s c t*) (`at` *level* 74 , *no associativity*) : *lang_cmd_scope*.

Structured commands (if-then-else's and while-loops) are parameterized by a type for boolean expressions.

File	Lines
SGoto in Coq (this document)	
while.v	458
goto.v	383
sgoto.v	689
sgoto_hoare.v	344
sgoto_hoare_example.v	374
compile.v	1177
compile_example.v	67
[AM06]	
mips_bipl.v	1222
mips_cmd.v	1001
mips_seplog.v	608

Table 2: Implementation Overview

```
Variable expr_b : Set.
Variable eval_b : expr_b → store → bool.
```

Using above types, we define the commands of WHILE languages.

```
Inductive cmd : Set :=
| cmd_cmd0 : cmd0 → cmd
| seq : cmd → cmd → cmd
| ifte : expr_b → cmd → cmd → cmd
| while : expr_b → cmd → cmd.
Coercion cmd_cmd0 : cmd0 >-> cmd.
Notation "c ; d" := (seq c d) (at level 81, right associativity) : lang_cmd_scope.
```

We now define the operational semantics of WHILE languages. Structured commands are given the textbook big-step operational semantics.

```
Reserved Notation "s – c —> t" (at level 74, no associativity).
Inductive exec : option state → cmd → option state → Prop :=
| exec_none : ∀ c, None – c —> None
| exec_cmd0 : ∀ s c s', s – c —-> s' → s – c —> s'
| exec_seq : ∀ s s' s" c d, s – c —> s' → s' – d —> s" → s – c ; d —> s"
| exec_ifte_true : ∀ s h s' t c d, eval_b t s → Some (s,h) – c —> s' →
  Some (s,h) – ifte t c d —> s'
| exec_ifte_false : ∀ s h s' t c d, ¬ eval_b t s → Some (s,h) – d —> s' →
  Some (s,h) – ifte t c d —> s'
| exec_while_true : ∀ s h s' s" t c, eval_b t s → Some (s,h) – c —> s' →
  s' – while t c —> s" → Some (s,h) – while t c —> s"
| exec_while_false : ∀ s h t c,
  ¬ eval_b t s → Some (s,h) – while t c —> Some (s,h)
where "s – c —> t" := (exec s c t) : lang_cmd_scope.
```

We now come to the formalization of textbook Hoare logic. Actually, we allow for an extension of Hoare logic with a notion of pointer and mutable memory (or heap for short) known as Separation logic. Assertions are shallow-encoded.

```
Let assert := store → heap → Prop.

Definition And (P Q : assert) : assert := fun s h ⇒ P s h ∧ Q s h.
Definition Not (P : assert) : assert := fun s h ⇒ ¬ P s h.
Definition entails (P Q : assert) : Prop := ∀ s h, P s h → Q s h.
Notation "P ===> Q" := (entails P Q) (at level 90, no associativity) : lang_cmd_scope.
```

The axioms of Hoare logic are encoded as an inductive type, assuming given Hoare triples for one-step, non-branching instructions.

```
Variable hoare0 : assert → cmd0 → assert → Prop.

Reserved Notation "{[ P ]} c {[ Q ]}" (at level 82, no associativity).
Inductive hoare : assert → cmd → assert → Prop :=
| hoare_hoare0 : ∀ P Q c, hoare0 P c Q → {[ P ]} c {[ Q ]}
| hoare_seq : ∀ P Q R c d, {[ P ]} c {[ Q ]} → {[ Q ]} d {[ R ]} → {[ P ]} c ; d {[ R ]}
| hoare_conseq : ∀ P P' Q Q' c, Q' ===> Q → P ===> P' →
  {[ P' ]} c {[ Q' ]} → {[ P ]} c {[ Q ]}
| hoare_while : ∀ P t c, {[ fun s h ⇒ P s h ∧ eval_b t s ]} c {[ P ]} →
  {[ P ]} while t c {[ fun s h ⇒ P s h ∧ ¬ eval_b t s ]}
| hoare_ifte : ∀ P Q t c d, {[ fun s h ⇒ P s h ∧ eval_b t s ]} c {[ Q ]} →
  {[ fun s h ⇒ P s h ∧ ¬ eval_b t s ]} d {[ Q ]} →
  {[ P ]} ifte t c d {[ Q ]}
where "{[ P ]} c {[ Q ]}" := (hoare P c Q) : lang_cmd_scope.

Definition hoare_semantics (P : assert) (c : cmd) (Q : assert) : Prop :=
  ∀ s h, P s h → ¬ Some (s,h) – c —> None ∧
    (∀ s' h', Some (s, h) – c —> Some (s', h') → Q s' h').

Definition wp_semantics (c : cmd) (Q : assert) : assert :=
  fun s h ⇒ ¬ (Some (s, h) – c —> None) ∧
    ∀ s' h', Some (s, h) – c —> Some (s', h') → Q s' h'.

End Lang.
```

2.2 Generic Properties of the Operational Semantics of While

We pack the generic syntax and the corresponding operational semantics above as a module:

```
Module Type WHILE_SEMOP.

Parameter store : Set.
Parameter heap : Type.
Definition state : Type := (store × heap)%type.

Parameter cmd0 : Set.
Parameter exec0 : option state → cmd0 → option state → Prop.
Notation "s – c —-> t" := (exec0 s c t) (at level 74 , no associativity) : goto_cmd_scope.
Parameter exec0_deter : ∀ (st : option state) (c : cmd0) (st' : option state),
  st – c —-> st' →
  ∀ st'', st – c —-> st'' → st' = st''.
Parameter from_none0 : ∀ (c : cmd0) s, None – c —-> s → s = None.
```

```
Parameter cmd0_terminate : ∀ (c : cmd0) s, ∃ s', Some s – c —-> s'.
```

```
Parameter expr_b : Set.
Parameter neg : expr_b → expr_b.
Parameter eval_b : expr_b → store → bool.
Parameter eval_b_neg : ∀ t s, ¬ eval_b t s ↔ eval_b (neg t) s.
Definition cmd := @cmd cmd0 expr_b.
Notation "c ; d" := (@seq cmd0 expr_b c d) (at level 81, right associativity) : goto_cmd_scope.
Coercion cmd_cmd0_coercion := @cmd_cmd0 cmd0 expr_b.
Definition exec := (@exec store heap cmd0 exec0 expr_b eval_b).
Notation "s – c —> t" := (exec s c t) (at level 74, no associativity) : goto_cmd_scope.
```

```
End WHILE_SEMOP.
```

We can derive some generic properties from the module above:

```
Module While_Semop_Prop (x : WHILE_SEMOP).
```

```
Import x.
```

```
Lemma from_none : ∀ c s, None – c —> s → s = None.
```

```
Lemma exec_deter : ∀ ST c ST', ST – c —> ST' →
  ∀ ST", ST – c —> ST" → ST' = ST".
```

```
End While_Semop_Prop.
```

2.3 Generic Properties of the Hoare Logic of While

We then pack the generic Hoare logic above as a module:

```
Module Type WHILE_HOARE.
```

```
Declare Module x : WHILE_SEMOP.
```

```
Import x.
```

```
Definition assert := store → heap → Prop.
Notation "P '//\\' Q" := (@And store heap P Q) (at level 80, no associativity) : goto_assert_scope.
Notation "P ===> Q" := (@entails store heap P Q) (at level 90, no associativity) : goto_assert_scope.
```

```
Parameter hoare0 : assert → cmd0 → assert → Prop.
```

```
Notation hoare_semantics := (@hoare_semantics store heap _ exec0 _ eval_b).
```

```
Parameter soundness0 : ∀ P Q c, hoare0 P c Q → hoare_semantics P c Q.
```

```
Definition hoare := @hoare store heap cmd0 _ eval_b hoare0.
```

```
Notation "{{ P }} c {{ Q }}" := (hoare P c Q) (at level 82, no associativity) : goto_hoare_scope.
```

```
Notation wp_semantics := (@wp_semantics store heap _ exec0 _ eval_b).
```

```
Parameter wp_semantics_sound0 : ∀ (c : cmd0) Q, {{ wp_semantics c Q }} c {{ Q }}.
```

The definition of Hoare logic for SGoto (Sect. 5) will require a function to compute the weakest precondition of one-step, non-branching instructions:

```
Parameter wp0 : cmd0 → assert → assert.
```

Parameter *wp0_no_err* : ∀ *s h c P*, *wp0 c P s h* → ¬ (*Some* (*s*,h) – *c* —-> *None*).
Parameter *exec0_wp0* : ∀ *s h* (*c* : *cmd0*) *s' h'*, *Some* (*s*, *h*) – *c* —-> *Some* (*s'*, *h'*) →
  ∀ (*P*:assert), *wp0 c P s h* ↔ *P s' h'*.

End *WHILE_HOARE*.

Finally, the Hoare logic must be shown to be sound and (relatively) complete, as capture by this last module:

Module *While_Hoare_Prop* (*x* : *WHILE_HOARE*).

Import *x*.
Import *x.x*.

Module *while_semop_prop_m* := *While_Semop_Prop x.x*.

Import *while_semop_prop_m*.

Lemma *soundness* : ∀ *P Q c*, {{ *P* }} *c* {{ *Q* }} → *hoare_semantics P c Q*.

Lemma *wp_semantics_sound*: ∀ *c Q*, {{ *wp_semantics c Q* }} *c* {{ *Q* }}.

Lemma *hoare_complete* : ∀ *P* Q *c*, *hoare_semantics P c Q* → {{ *P* }} *c* {{ *Q* }}.

End *While_Hoare_Prop*.

3 Goto: A Low-level Language

This section corresponds to Section 2 in [SU07].

Module *Goto* (*x* : *while.WHILE_SEMOP*).

Import *x*.

3.1 Syntax and (Small-step) Semantics of Goto

Definition label := **nat**.

Definition lstate := **option** (label × state).

For the operational semantics of one-step, non-branching instructions of Goto, we use the one-step commands (type *cmd0* and operational semantics noted · – · → ·) (see Section 2).

Reserved Notation " c ⊢ s → t " (at *level* 82, *no associativity*).
Inductive **exec0_label** : lstate → *cmd0* → lstate → Prop :=
| exec0_label_cmd0 :
  ∀ *s c s'*, Some *s* – *c* → Some *s'* → ∀ *l*, **exec0_label** (Some (*l*, *s*)) *c* (Some (S *l*, *s'*))
| exec0_label_err :
  ∀ *s c*, Some *s* – *c* → None → ∀ *l*, **exec0_label** (Some (*l*, *s*)) *c* None
where " c ⊢ s → t " := (**exec0_label** *s c t*) : *sgoto_scope*.

Branches may be conditional or not. For conditional branches, we use a language of boolean expressions (type *expr_b*) (see Section 2):

```
Inductive branch : Set := jmp : label → branch | cjmp : expr_b → label → branch.
```

Note that branches never cause errors:

```
Inductive exec_branch : label × state → branch → label × state → Prop :=
| exec_jmp : ∀ p s l, jmp l ⊢ (p, s) ≫ (l, s)
| exec_cjmp_true : ∀ p s h t l, eval_b t s → cjmp t l ⊢ (p, (s, h)) ≫ (l, (s, h))
| exec_cjmp_false : ∀ p s h t l, ¬ eval_b t s → cjmp t l ⊢ (p,(s,h)) ≫ (S p, (s, h))
where "c ⊢ s ≫ t" := (exec_branch s c t) : sgoto_scope.
```

Unstructured programs are lists of labeled (branching or not) instructions. They are wellformed when no instruction has two labels:

```
Inductive insn : Set := C : cmd0 → insn | B : branch → insn.

Definition code := list (label × insn).

Definition wellformed_goto (c:code) : Prop := ∀ l i i', In (l,i) c → In (l,i') c → i = i'.
```

We can now define the semantics of Goto. The type below corresponds to Figure 1 (Small-step semantics rules of Goto) in [SU07]:

```
Inductive exec_goto : code → lstate → lstate → Prop :=
| exec_goto_cmd0 : ∀ p i s s' c,
  In (p, C i) c → i ⊢ Some (p, s) → Some s' → c ⊢ Some (p, s) ↠ Some s'
| exec_goto_cmd0_err : ∀ p i s c,
  In (p, C i) c → i ⊢ Some (p, s) → None → c ⊢ Some (p, s) ↠ None
| exec_goto_branch : ∀ p j s s' c,
  In (p, B j) c → j ⊢ (p, s) ≫ s' → c ⊢ Some (p, s) ↠ Some s'
where "c ⊢ s ↠ t" := (exec_goto c s t) : sgoto_scope.
```

3.2 Properties

Lemma 1 (**Determinacy**) in [SU07]:

```
Lemma exec_goto_deter : ∀ c, wellformed_goto c →
 ∀ s s', c ⊢ s ↠ s' → ∀ s", c ⊢ s ↠ s" → s' = s".
```

See the end of Section 3.3 for a comment about Lemma 2 (**Stuck states**).
Lemma 3 (**Extension of the domain**) in [SU07]:

```
Lemma exec_goto_extension_right : ∀ c' s s' c, c ⊢ s ↠ s' → c ++ c' ⊢ s ↠ s'.

Lemma exec_goto_contraction_right : ∀ c1 c2, wellformed_goto (c1 ++ c2) →
  ∀ l s l' s', c1 ++ c2 ⊢ Some (l,s) ↠ Some (l',s') →
    In l (dom c1) → c1 ⊢ Some (l,s) ↠ Some (l',s').

Lemma exec_goto_extension_left : ∀ c s s' i, c ⊢ s ↠ s' → i :: c ⊢ s ↠ s'.

Lemma exec_goto_contraction_left : ∀ c1 c2, wellformed_goto (c1 ++ c2) →
 ∀ l s l' s', c1 ++ c2 ⊢ Some (l,s) ↠ Some (l', s') →
  In l (dom c2) → c2 ⊢ Some (l, s) ↠ Some (l', s').
```

3.3 Reflexive, Transitive Closure Predicates

Reflexive, transitive closure, to be used in Theorem 6 (**Preservation of evaluations as stuck reduction sequences**) of [SU07]:

`Inductive` **redseqs** : code → lstate → lstate → `Prop` :=
| redseqs_refl : ∀ s c, $c \vdash s \twoheadrightarrow^* s$
| redseqs_trans : ∀ s s' s'' c, $c \vdash s \twoheadrightarrow^* s' \to c \vdash s' \twoheadrightarrow s'' \to c \vdash s \twoheadrightarrow^* s''$
`where` " c ⊢ s '↠*' t " := (**redseqs** c s t) : *sgoto_scope*.

Reflexive, transitive closure with explicit index k, to be used in Theorem 7 (**Reflection of stuck reduction sequences as evaluations**):

`Inductive` **redseq** (p : code) : **nat** → lstate → lstate → `Prop` :=
| zero_red : ∀ s, **redseq** p O s s
| more_red : ∀ n s s' s'', $p \vdash s \twoheadrightarrow s' \to$ **redseq** p n s' $s'' \to$ **redseq** p (S n) s s''.

The following two lemmas express, in the particular case of branches, a property similar to Lemma 2 (**Stuck states**) in [SU07]. They are used in the proof of Theorem 7 (**Reflection of stuck reduction sequences as evaluations**) in lieu of Lemma 2.

Lemma redseq_out_of_domain_jump : ∀ k p m l st l' st', $p \neq l \to$
  redseq ((p, B (jmp m)) :: nil) k (Some (l, st)) (Some (l', st')) → $l = l' \land st = st'$.

Lemma redseq_out_of_domain_cjmp : ∀ k p t m l st l' st', $p \neq l \to$
  redseq ((p, B (cjmp t m))::nil) k (Some (l, st)) (Some (l', st')) → $l = l' \land st = st'$.

`End` Goto.

4 SGoto, A Structured Version

This corresponds to Section 3.1 of [SU07].

`Module` *SGoto* (x : *while.WHILE_SEMOP*).

`Module` *goto_m* := *Goto x*.
`Import` *goto_m*.
`Import` x.

4.1 Natural Semantics Rules of SGoto

`Inductive` **scode** : `Set` :=
| sO : **scode**
| sC : label → *cmd0* → **scode**
| sB : label → **branch** → **scode**
| sS : **scode** → **scode** → **scode**.

`Notation` "c '⊕' d" := (sS c d) (at *level* 69, *right associativity*) : *sgoto_scope*.

`Fixpoint` sdom sc :=
  `match` sc `with`

```
    | sO ⇒ nil | sC l _ ⇒ l :: nil | sB l _ ⇒ l :: nil
    | sc1 [+] sc2 ⇒ sdom sc1 ++ sdom sc2
  end.
```

Structured code is wellformed when instructions all have different labels:

```
Inductive wellformed : scode → Prop :=
| wf_sO : wellformed sO
| wf_sC : ∀ x y, wellformed (sC x y)
| wf_sB : ∀ x y, wellformed (sB x y)
| wf_sS : ∀ sc1 sc2, inter (sdom sc1) (sdom sc2) nil →
  wellformed sc1 → wellformed sc2 → wellformed (sc1 [+] sc2).
```

The forgetful function forgets the structure of the code, effectively turning a piece of SGoto code into a piece of Goto code:

```
Fixpoint U sc :=
  match sc with
    | sO ⇒ nil | sC l c ⇒ (l, C c) :: nil | sB l b ⇒ (l, B b) :: nil
    | sc1 [+] sc2 ⇒ U sc1 ++ U sc2
  end.
```

We can now define the semantics of SGoto. The inductive type below corresponds to Figure 2 (Natural semantics rules of SGoto) in [SU07]. Note that there is an additional constructor for error propagation.

```
Inductive exec_sgoto : scode → lstate → lstate → Prop :=
| exec_sgoto_none : ∀ c, None ≻ c → None
| exec_sgoto_cmd0 : ∀ p c st s', c ⊢ Some (p, st) → s' → Some (p, st) ≻ sC p c → s'
| exec_sgoto_jmp : ∀ p st p', p ≠ p' → Some (p, st) ≻ sB p (jmp p') → Some (p', st)
| exec_sgoto_cjmp_true : ∀ p s h b p',
  eval_b b s → p ≠ p' → Some (p, (s,h)) ≻ sB p (cjmp b p') → Some (p', (s,h))
| exec_sgoto_cjmp_false : ∀ p s h b p',
  ¬ eval_b b s → Some (p, (s,h)) ≻ sB p (cjmp b p') → Some (S p, (s,h))
| exec_sgoto_seq0 : ∀ sc1 sc2 p st s' s", In p (sdom sc1) → Some (p, st) ≻ sc1 → s' →
  s' ≻ sc1 [+] sc2 → s" → Some (p, st) ≻ sc1 [+] sc2 → s"
| exec_sgoto_seq1 : ∀ sc1 sc2 p st s' s", In p (sdom sc2) → Some (p, st) ≻ sc2 → s' →
  s' ≻ sc1 [+] sc2 → s" → Some (p, st) ≻ sc1 [+] sc2 → s"
| exec_sgoto_refl : ∀ sc p st, ¬ In p (sdom sc) → Some (p, st) ≻ sc → Some (p, st)
where "s ≻ p → t" := (exec_sgoto p s t) : sgoto_scope.
```

4.2 Properties

Lemma 4 (**Determinacy**) in [SU07]:

```
Lemma determinacy : ∀ c (Hwf: wellformed c), ∀ s s', s ≻ c → s' → ∀ s", s ≻ c → s" → s' = s".
```

Lemma 5 (**Postlabels**) in [SU07]:

```
Lemma postlabels : ∀ c s l' st', s ≻ c → Some (l',st') → ¬ In l' (sdom c).
```

Theorem 6 (**Preservation of evaluations as stuck reduction sequences**) in [SU07].

Lemma preservation : ∀ *prg s s'*, *s* ≻ *prg* → *s'* → U *prg* ⊢ *s* ↠* *s'*.

Theorem 7 (**Reflection of stuck reduction sequences as evaluations**) in [SU07]. Nested induction whose inner induction is noetherian.

```
Require Import Wf_nat.
```

Lemma reflection_of_stuck_redseq : ∀ *prg k l st l' st'* (*Hwf* : wellformed_goto (U *prg*)),
  redseq (U *prg*) *k* (Some (*l*, *st*)) (Some (*l'*, *st'*)) →
  ¬ In *l'* (sdom *prg*) →
  Some (*l*, *st*) ≻ *prg* → Some (*l'*,st').

4.3 Semantic Equivalence

Definition sem_equ *sc0 sc1* := ∀ *s s'*, Some *s* ≻ *sc0* → Some *s'* ↔ Some *s* ≻ *sc1* → Some *s'*.

Notation "c '≅' d" := (sem_equ *c d*) (at *level* 70, *right associativity*) : *sgoto_scope*.

Theorem 8 (**Neutrality wrt phrase structure**) in [SU07]:

Lemma neutrality : ∀ *sc0 sc1*, **wellformed** *sc0* → U *sc0* = U *sc1* →
  ∀ *s s'*, Some *s* ≻ *sc0* → Some *s'* →
   Some *s* ≻ *sc1* → Some *s'*.

Corollary 9 (**Partial commutative monoidal structure**) in [SU07].

Lemma sem_equ_ass : ∀ *sc0 sc1 sc2*, **wellformed** ((*sc0* [+] *sc1*) [+] *sc2*) →
  (*sc0* [+] *sc1*) [+] *sc2* ≅ *sc0* [+] (*sc1* [+] *sc2*).

Lemma sem_equ_neu : ∀ *sc*, **wellformed** *sc* → *sc* [+] sO ≅ *sc*.

Interestingly, commutativity does not require well-formedness:

Lemma sem_equ_com : ∀ *sc0 sc1*, *sc0* [+] *sc1* ≅ *sc1* [+] *sc0*.

End SGoto.

5 Hoare Logic of SGoto

This corresponds to Section 3.2 of [SU07]. The type `assert` was defined in Section 2.

Module *SGoto_Hoare* (*x* : *while.WHILE_HOARE*).

Module *sgoto_m* := *SGoto x.x*.
Import *sgoto_m*.
Import *goto_m*.
Import *x*.
Import *x.x*.

Definition *assn* := *label* → `assert`.

Local Open Scope *goto_assert_scope*.

Definition *restrict* (*P* : *assn*) *d* : *assn* := `fun` *l* ⇒ *P l* ∧ (`fun` _ _ ⇒ *In l d*).

Definition *restrict_cplt* (P : *assn*) d : *assn* := fun $l \Rightarrow$ *while.Not* (fun _ _ $\Rightarrow$ *In* l d) $\wedge$ P l.

Figure 3 (Hoare rules of SGoto) in [SU07]. *wp0* is explained in Section 2. $\Longrightarrow$ used in the rule *hoare_sgoto_conseq* is the entailment for assert.

Notation "'_assn'" := *assn* : *sgoto_hoare_scope*.

Local Open Scope *sgoto_scope*.
Local Open Scope *sgoto_hoare_scope*.

Inductive *hoare_sgoto* : *assn* $\to$ *scode* $\to$ *assn* $\to$ Prop :=
| *hoare_cmd* : $\forall$ l c P,
  [^ fun $pc \Rightarrow$ fun s $h \Rightarrow pc = l \wedge (wp0\ c\ (P\ (S\ l)))\ s\ h \vee pc \neq l \wedge P\ pc\ s\ h$ ^]
    sC l c [^ P ^]
| *hoare_jmp* : $\forall$ l j Q,
  [^ fun $pc \Rightarrow$ fun s $h \Rightarrow pc = l \wedge (Q\ j\ s\ h \vee j = l) \vee pc \neq l \wedge Q\ pc\ s\ h$ ^]
    sB l (jmp j) [^ Q ^]
| *hoare_branch* : $\forall$ l b j Q,
  [^ fun $pc \Rightarrow$ fun s $h \Rightarrow$
    $pc = l \wedge (\neg\ eval\_b\ b\ s \wedge Q\ (S\ l)\ s\ h \vee eval\_b\ b\ s \wedge (\ Q\ j\ s\ h \vee j = l)) \vee$
    $pc \neq l \wedge Q\ pc\ s\ h$ ^]
    sB l ($cjmp$ b j) [^ Q ^]
| *hoare_sO* : $\forall$ P, [^ P ^] sO [^ P ^]
| *hoare_sS* : $\forall$ $sc0$ $sc1$ P,
  [^ *restrict* P (*sdom* $sc0$) ^] $sc0$ [^ P ^] $\to$ [^ *restrict* P (*sdom* $sc1$) ^] $sc1$ [^ P ^] $\to$
  [^ P ^] $sc0$ [+] $sc1$ [^ *restrict_cplt* P (*sdom* ($sc0$ [+] $sc1$)) ^]
| *hoare_sgoto_conseq* : $\forall$ sc (P Q P' Q': *assn*),
  ($\forall$ l, $P\ l \Longrightarrow P'\ l$) $\to$ ($\forall$ l, $Q'\ l \Longrightarrow Q\ l$) $\to$
  [^ P' ^] sc [^ Q' ^] $\to$ [^ P ^] sc [^ Q ^]
where "'[^' P '^]' c '[^' Q '^]'" := (*hoare_sgoto* P c Q) : *sgoto_hoare_scope*.

Theorem 10 (**Soundness**) in [SU07]:

Module *while_semop_prop_m* := *while.While_Semop_Prop x.x*.

Lemma *hoare_sgoto_sound* : $\forall$ sc P Q, [^ P ^] sc [^ Q ^] $\to$
  $\forall$ l s h, $P\ l\ s\ h \to$
    $\neg$ (*Some* (l, (s, h)) $\succ sc \to$ *None*) $\wedge$
    $\forall$ l' s' h', *Some* (l, (s, h)) $\succ sc \to$ *Some* (l', (s', h')) $\to Q\ l'\ s'\ h'$.

The semantic definition of the weakest precondition from [SU07]. The additional conjunct is to take errors into account.

Definition *wlp_semantics* (sc: *scode*) (Pi: *assn*) : *assn* := fun $l \Rightarrow$ fun s $h \Rightarrow$
  $\neg$ (*Some* (l, (s, h)) $\succ sc \to$ *None*) $\wedge$
  $\forall$ l' s' h', *Some* (l, (s, h)) $\succ sc \to$ *Some* (l', (s', h')) $\to Pi\ l'\ s'\ h'$.

Lemma 11 in [SU07]:

Lemma *wlp_completeness* : $\forall$ sc (Hwf: *wellformed* sc) Q, [^ *wlp_semantics* sc Q ^] sc [^ Q ^].

Theorem 12 (**Completeness**) in [SU07].

Lemma *hoare_sgoto_complete* : $\forall$ (P Q: *assn*) sc (Hwf: *wellformed* sc),

```
(∀ l s h,
   P l s h →
   ¬ ( Some (l, (s, h)) ≻ sc → None ) ∧
   (∀ l' s' h', Some (l,(s,h)) ≻ sc → Some (l',(s',h')) → Q l' s' h')) →
[^ P ^] sc [^ Q ^].
End SGoto_Hoare.
```

6 Example: The Sum of the n First Naturals

This example corresponds to Section 4.3 in [SU07]. The main difference is that the program is shown to compute its result *modulo* 2^{32}, which is not the case with the archetypal assembly language of [SU07].

We first define registers to hold an intermediate value x, the output r, and the input n. Since registers have a finite size, the number of values that can be represented is limited.

```
Definition x := reg_t0.
Definition r := reg_t1.
Definition n := reg_t2.
```

The program consists of the following four labeled instructions:

```
Definition i1 := sB 1 (cjmp (beq x n) 5).
Definition i2 := sC 2 (addiu x x 1_16).
Definition i3 := sC 3 (addu r x r).
Definition i4 := sB 4 (jmp 1).
Definition prg : scode := i1 [+] ((i2 [+] i3) [+] i4).
```

The pre-condition is as follows. The output value r is initialized to 0 and the input value is expected to be positive (which actually holds naturally when registers' contents are regarded as unsigned).

```
Definition I1 : assn := fun pc ⇒ fun s h ⇒ pc = 1 ∧ 0_32 [.≤] [n]_s ∧ [x]_s = 0_32 ∧ [r]_s = 0_32.
```

The post-condition is as follows. The intermediate value x (repeatedly incremented during execution) is expected to be equal to the input value n and the output value is exepected to be equal to the sum of the n first naturals *modulo* 2^{32}. The non-modulo equality cannot be achieved in practice because of potential arithmetic overflows. u2Z is a function that interprets a finite-size integer as unsigned and returns its decimal value.

```
Local Open Scope zarith_ext_scope.
```

```
Definition I5' : assn := fun pc ⇒ fun s h ⇒ pc = 5 ∧
  [x]_s = [n]_s ∧ u2Z [r]_s = Zsum (u2Z [x]_s) {{2^^32}}.
```

The correctness proof consists of the application of the rules of the Hoare logic for SGoto. For the purpose of presentation, this proof can be decomposed in a sequence of basic steps, each consisting of the application of a single rule of the Hoare logic. For example, the following step shows that the addition of the intermediate value really corresponds to compute and add the next natural.

Definition $I2'$: $assn$:= fun $pc \Rightarrow$ fun $s\ h \Rightarrow pc = 2 \wedge$
$[x]\_s\ [.\!<]\ [n]\_s \wedge u2Z\ [r]\_s = Zsum\ (u2Z\ [x]\_s)\ \{\{\ 2\hat{}\hat{}32\ \}\}$.

Definition $I2''$: $assn$:= fun $pc \Rightarrow$ fun $s\ h \Rightarrow pc = 2 \wedge$
$[x]\_s\ [.+]\ 1_{32}\ [.\!\leq]\ [n]\_s \wedge$
$u2Z\ [r]\_s + u2Z\ ([x]\_s\ [.+]\ 1_{32}) = Zsum\ (u2Z\ ([x]\_s\ [.+]\ 1_{32}))\ \{\{2\hat{}\hat{}32\}\}$.

Definition $I3$: $assn$:= fun $pc \Rightarrow$ fun $s\ h \Rightarrow pc = 3 \wedge [x]\_s\ [.\!\leq]\ [n]\_s \wedge$
$u2Z\ ([x]\_s\ [.+]\ [r]\_s) = Zsum\ (u2Z\ [x]\_s)\ \{\{2\hat{}\hat{}32\}\}$.

Lemma *step_18* : $[[\ I2''\ ]]\ i2\ [[\ I3\ ]] \rightarrow [[\ I2'\ ]]\ i2\ [[\ I3\ ]]$.

Once all such steps are proved individually, the correctness proof consists in the sequential application of the corresponding lemmas:

Lemma *prf* : $[[\ I1\ ]]\ prg\ [[\ I5'\ ]]$.
apply *step_1*.
apply *step_2*.
apply *step_3*.
apply *step_4*.
apply *step_5*.
apply *step_6*.
apply *step_7*; *last first*.
apply *step_8*.
apply *step_9*.
apply *step_10*.
apply *step_11*.
apply *step_12*.
apply *step_13*; *last first*.
apply *step_14*.
apply *step_15*.
apply *step_16*.
apply *step_17*.
apply *step_18*.
apply *step_19*.
apply *step_20*.
Qed.

Module COMPILE (x : WHILE.WHILE_HOARE).

Module SGOTO_HOARE_M := SGOTO_HOARE X.
Import *sgoto_hoare_m*.
Import *sgoto_m*.
Import *goto_m*.
Import *x*.
Import *x.x*.

Module WHILE_PROP_M := WHILE.WHILE_SEMOP_PROP X.X.

7 Compilation from While to SGoto

This corresponds to Section 4 of [SU07].

7.1 Compilation and Preservation/Reflection of Evaluations

Figure 5 (Rules of compilation from While to SGoto) in [SU07]. A slight difference is that we do not remove nop instructions (they are sometimes important in MIPS assembly because of non-taken branch prediction).

```
Import while.
Inductive compile : label → @cmd cmd0 expr_b → scode → label → Prop :=
| comp_cmd : ∀ l (c : cmd0) , compile l c (sC l c) (S l)
| comp_seq : ∀ l l' l'' c d c' d',
  compile l c c' l'' → compile l'' d d' l' → compile l (c ; d) (c' [+] d') l'
| comp_ifte : ∀ l l' l'' t c d c' d',
  compile (S l'') c c' l' → compile (S l) d d' l'' →
  compile l (ifte t c d) (sB l (cjmp t (S l''')) [+] ((d' [+] sB l'' (jmp l')) [+] c')) l'
| comp_while : ∀ l l' t c prg,
  compile (S l) c prg l' →
  compile l (while t c) (sB l (cjmp (neg t) (S l'))) [+] (prg [+] sB l' (jmp l))) (S l').
```

Lemma 13 (**Totality and determinacy of compilation**) in [SU07]:

```
Lemma totality : ∀ l c, ∃ sc, ∃ l', compile l c sc l'.
Lemma determinacy : ∀ c l l'0 sc0, compile l c sc0 l'0 →
  ∀ l'1 sc1, compile l c sc1 l'1 →
    sc0 = sc1 ∧ l'0 = l'1.
```

Lemma 14 (**Domain of compiled code**) in [SU07]:

```
Lemma compile_sdom : ∀ c l sc l', compile l c sc l' → ∀ p, l ≤ p < l' → In p (sdom sc).
Lemma compile_sdom' : ∀ c l sc l', compile l c sc l' → ∀ p, In p (sdom sc) → l ≤ p < l'.
```

Compilation always produces wellformed code:

```
Lemma compile_wellformed : ∀ c l sc l', compile l c sc l' → wellformed sc.
```

Theorem 15 (**Preservation of evaluations**) in [SU07]:

```
Lemma preservation_of_evaluations : ∀ c s l sc s' l',
 compile l c sc l' →
  Some s – c → Some s' →
   Some (l, s) ≻ sc → Some (l + length (sdom sc), s').
```

Theorem 16 (**Reflection of evaluations**) in [SU07].

This proof is done by a nested induction to handle the while-case. We isolate this subcase by intermediate lemmas (one lemma for the error-free case and another lemma for the error case). Here follows the intermediate lemma for the error-free case; what will be the outer induction hypothesis in the main proof is given as an hypothesis to this intermediate lemma.

```
Lemma reflection_of_evaluations' : ∀ c_t
  (IHouter : ∀ l sc_t l' s s' lstar, compile l c_t sc_t l' →
    Some (l, s) ≻ sc_t → Some (lstar, s') →
    lstar = l' ∧ (Some s – c_t → Some s')) sc st st',
  st ≻ sc → st' →
  ∀ l l' t, compile l (while t c_t) sc l' →
    ∀ s h lstar s' L,
      L = l ∨ L = S l →
      ∀ (Hneq: eval_b t s),
        st = Some (L, (s, h)) →
        st' = Some (lstar, s') →
        lstar = l' ∧ (Some (s, h) – while t c_t → Some s').
```

```
Lemma reflection_of_evaluations: ∀ c l sc l', compile l c sc l' →
  ∀ s, (∀ lstar s',
    Some (l, s) ≻ sc → Some (lstar, s') → lstar = l' ∧ (Some s – c → Some s')) ∧
  (Some (l, s) ≻ sc → None → (Some s – c → None)).
```

7.2 Preservation/Reflection of Derivable Hoare Triples

Theorem 17 (**Preservation of derivable Hoare triples**) in [SU07]. The proof of this theorem makes use of the soundness of Hoare logic for WHILE; this is the lemma *soundness* used below.

```
Module WHILE_HOARE_PROP_M := WHILE_HOARE_PROP X.
```

```
Lemma preservation_hoare :
 ∀ P Q c, {{ P }} c {{ Q }} →
 ∀ l sc l', compile l c sc l' →
 [^ fun pc ⇒ fun s h ⇒ pc = l ∧ P s h ^] sc [^ fun pc ⇒ fun s h ⇒ pc = l' ∧ Q s h ^].
Proof.
move⇒ P Q c Hoare l sc l' Hcompile.
apply hoare_sgoto_complete; first by eapply compile_wellformed; eauto.
move⇒ l0 s h [→ HP] {l0}.
move/while_hoare_prop_m.soundness: Hoare.
case/(__ _ HP) ⇒ Herror_free HQ.
move/reflection_of_evaluations: Hcompile.
case/(_(s, h)) ⇒ Hcompile1 Hcompile2.
split.
- by move⇒ X; apply Hcompile2 in X.
- move⇒ l'_ s' h' Hexec.
  case/Hcompile1 : Hexec ⇒ Hl'_l'.
  by move/HQ.
Qed.
```

Theorem 18 (**Reflection of derivable Hoare triples**). The proof of this theorem uses in particular the completeness of Hoare-logic for WHILE.

```
Lemma reflection_hoare : ∀ l c sc l', compile l c sc l' →
  ∀ P Q, [^ P ^] sc [^ Q ^] → {{ P l }} c {{ Q l' }}.
```

End COMPILE.

8 Application: Generation of Hoare-logic Proofs from WHILE

As explained in Section 1, in [AM06], we verified in Coq an implementation of the Montgomery multiplication written in the SmartMIPS instruction set. We worked on a version of the program where branches were replaced by while-loops and while-loops where compiled away by a certified macro-expander afterwards. Strictly speaking, there was therefore no Hoare-logic proof for the assembly code to be run.

The rest of this section shows that one can recover a Hoare-logic proof for the assembly code to be run by using the previously formalized theorem *preservation_hoare* (Section 7.2).

montgomery is the program with while-loops. We instantiate it with a set of registers:

`Definition` *mont_mul_cmd* : *while.cmd* := *montgomery* k *alpha x y z m one ext int_ X_ Y_* M_ *Z_ quot* C *t s_*.

Given a certain set of parameters (concrete initial values to put in registers and in the mutable memory), the proof of correctness *mont_mul_specif* gives a proof-term that is the proof that the Montgomery multiplication with while-loops is correct. In other words, this is a proof of correctness prior to compilation. This is clear when checked with the `Check` command.

`Definition` *mont_mul_cmd_hoare* :=
mont_mul_triple _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Hset nk valpha nx ny nm nz vx vy vm vz X Y *M Halpha* Hx *Hy Hm Hnz Hvx* Hvy *Hvm Hvz HX* HY.

`Check` *mont_mul_cmd_hoare*.

```
> Check mont_mul_cmd_hoare.
{{fun s h => [x]_s = vx /\ [y]_s = vy /\ [z]_s = vz /\ [m]_s = vm /\
  u2Z ([k]_s) = Z_of_nat nk /\ [alpha]_s = valpha /\
  (((var_e x |--> X ** var_e y |--> Y) ** var_e z |--> Lists_ext.rep zero32 nk) **
   var_e m |--> M) s h /\
  store.multi_null s}}
montgomery k alpha x y z m one ext int_ X_ Y_ M_ Z_ quot C t s_
{{fun s h => exists Z0, length Z0 = nk /\
  [x]_s = vx /\ [y]_s = vy /\ [z]_s = vz /\ [m]_s = vm /\
  u2Z ([k]_s) = Z_of_nat nk /\ [alpha]_s = valpha /\
  (((var_e x |--> X ** var_e y |--> Y) ** var_e z |--> Z0) ** var_e m |--> M) s h /\
  (Zbeta nk * Sum nk.+1 (Z0 ++ [C]_s :: nil) =m Sum nk X * Sum nk Y {{Sum nk M}}) /\
  Sum nk.+1 (Z0 ++ [C]_s :: nil) < 2 * Sum nk M /\
  u2Z ([t]_s) = 4 * nz + 4 * Z_of_nat (nk - 1)}}
```

Now, let us consider *mont_mul_scode*, the Montgomery multiplication with gotos, obtained by automatically macro-expanding if-then-else's and while-loops and locating the code at starting label 0 (using a function corresponding to the *compile* predicate (see Section 7.1)):

`Definition` *mont_mul_scode* : *compile_m.sgoto_hoare_m.sgoto_m.scode* := *compile_m.compile_f O mont_mul_cmd*.

By application of *preservation_hoare* and given the proof that the Montgomery multiplication with while-loops is correct, we obtain a proof-term that is the proof that the Montgomery multiplication *with gotos* is correct. Again, this can be checked with the `Check` command: the same triple as above is shown to hold, with the additional information that the starting label is 0, and the ending label is 38.

`Definition` *mont_mul_sgoto_hoare* :=
  compile_m.preservation_hoare _ _ _ *mont_mul_cmd_hoare* _ _ _ *Hcompile.*

```
> Check mont_mul_sgoto_hoare.
compile_m.sgoto_hoare_m.hoare_sgoto
(fun pc s h0 => pc = /\ (fun s0 h =>
 [x]_s0 = vx /\ [y]_s0 = vy /\ [z]_s0 = vz /\ [m]_s0 = vm /\
 u2Z ([k]_s0) = Z_of_nat nk /\ [alpha]_s0 = valpha /\
 (((var_e x |--> X ** var_e y |--> Y) ** var_e z |--> Lists_ext.rep zero32 nk) **
  var_e m |--> M) s0 h /\
 store.multi_null s0) s h0)
mont_mul_scode
(fun pc s h0 => pc = 38 /\ (fun s0 h => exists Z0, length Z0 = nk /\
 [x]_s0 = vx /\ [y]_s0 = vy /\ [z]_s0 = vz /\ [m]_s0 = vm /\
 u2Z ([k]_s0) = Z_of_nat nk /\ [alpha]_s0 = valpha /\
 (((var_e x |--> X ** var_e y |--> Y) ** var_e z |--> Z0) ** var_e m |--> M) s0 h /\
 (Zbeta nk * Sum nk.+1 (Z0 ++ [C]_s0 :: nil) =m Sum nk X * Sum nk Y {{Sum nk M}}) /\
 Sum nk.+1 (Z0 ++ [C]_s0 :: nil) < 2 * Sum nk M /\
 u2Z ([t]_s0) = 4 * nz + 4 * Z_of_nat (nk - 1)) s h0)
```

References

[AM06] Reynald Affeldt and Nicolas Marti. An approach to formal verification of arithmetic functions in assembly. In *11th Annual Asian Computing Science Conference (ASIAN 2006), Focusing on Secure Software and Related Issues*, volume 4435 of *LNCS*, pages 346–360. Springer, 2006.

[GM07] Georges Gonthier and Assia Mahboubi. A small scale reflection extension for the Coq system. Technical Report 6455, INRIA, Dec. 2007.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002. Invited lecture.

[SU07] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theor. Comput. Sci.*, 373(3):273–302, 2007. Elsevier.