Formal verification of hardware synthesis

Thomas Braibant   Adam Chlipala

University of Grenoble

MIT CSAIL

Coq Workshop 2012

Context

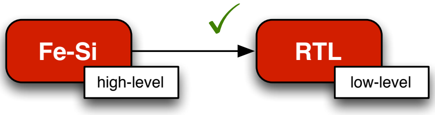
- Formally verified everything:
  - Compilers (CompCert [2006])
  - Operating Systems (Gypsy [1989]; seL4 [2009])
  - Static analysers
  - Hardware
- Verifying hardware with theorem provers:
  - many *shallow-embeddings* of hardware description languages (ACL2 , HOL, PVS)
  - many *shallow-embeddings* of hardware designs (ACL2, HOL, PVS, Coq)
    - Floating-point operations verified at AMD using ACL2
    - VAMP [2003] (a pipelined micro-processor verified in PVS)
  - ...
- Industry shifts toward hardware synthesis:
  - generates low-level code (RTL) from high-level HDLs
  - argue (in)formally that this synthesis is correct

- Formally verified everything:
  - Compilers (CompCert [2006])
  - Operating Systems (Gypsy [1989]; seL4 [2009])
  - Static analysers
  - Hardware
- Verifying hardware with theorem provers:
  - many *shallow-embeddings* of hardware description languages (ACL2 , HOL, PVS)
  - many *shallow-embeddings* of hardware designs (ACL2, HOL, PVS, Coq)
    - Floating-point operations verified at AMD using ACL2
    - VAMP [2003] (a pipelined micro-processor verified in PVS)
  - ...
- Industry shifts toward hardware synthesis:
  - generates low-level code (RTL) from high-level HDLs
  - argue (in)formally that this synthesis is correct

Context

- Formally verified everything:
  - Compilers (CompCert [2006])
  - Operating Systems (Gypsy [1989]; seL4 [2009])
  - Static analysers
  - Hardware
- Verifying hardware with theorem provers:
  - many *shallow-embeddings* of hardware description languages (ACL2 , HOL, PVS)
  - many *shallow-embeddings* of hardware designs (ACL2, HOL, PVS, Coq)
    - Floating-point operations verified at AMD using ACL2
    - VAMP [2003] (a pipelined micro-processor verified in PVS)
  - ...
- Industry shifts toward hardware synthesis:
  - generates low-level code (RTL) from high-level HDLs
  - argue (in)formally that this synthesis is correct

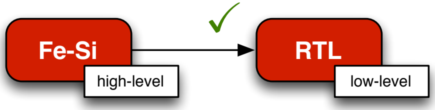
- Investigate hardware synthesis in Coq



- Source language: Fe-Si (Featherweight Synthesis)
  - Stripped down and simplified version of Bluespec
  - Semantics based on “guarded atomic actions” (with a flavour of transactional memory)
- Target language: RTL
  - Combinational logic and next-state assignments for registers
  - No currents, no delays, single-clock
- We define *deep-embeddings*
  - Define data-structures to represent programs
  - Define what is a program’s semantics (via an interpretation function)
  - Use parametric higher-order abstract syntax (PHOAS) to deal with binders

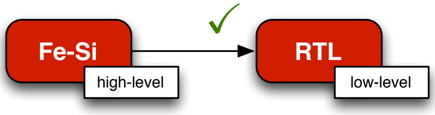
This project
on-going work

- Investigate hardware synthesis in Coq



- Source language: Fe-Si (Featherweight Synthesis)
  - Stripped down and simplified version of Bluespec
  - Semantics based on “guarded atomic actions” (with a flavour of transactional memory)
- Target language: RTL
  - Combinational logic and next-state assignments for registers
  - No currents, no delays, single-clock
- We define *deep-embeddings*
  - Define data-structures to represent programs
  - Define what is a program’s semantics (via an interpretation function)
  - Use parametric higher-order abstract syntax (PHOAS) to deal with binders

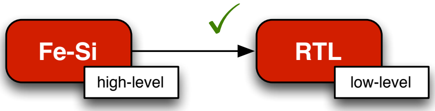
- Investigate hardware synthesis in Coq



- Source language: Fe-Si (Featherweight Synthesis)
  - Stripped down and simplified version of Bluespec
  - Semantics based on “guarded atomic actions” (with a flavour of transactional memory)
- Target language: RTL
  - Combinational logic and next-state assignments for registers
  - No currents, no delays, single-clock
- We define *deep-embeddings*
  - Define data-structures to represent programs
  - Define what is a program’s semantics (via an interpretation function)
  - Use parametric higher-order abstract syntax (PHOAS) to deal with binders

This project
on-going work

- Investigate hardware synthesis in Coq



- Source language: Fe-Si (Featherweight Synthesis)
  - Stripped down and simplified version of Bluespec
  - Semantics based on “guarded atomic actions” (with a flavour of transactional memory)
- Target language: RTL
  - Combinational logic and next-state assignments for registers
  - No currents, no delays, single-clock
- We define *deep-embeddings*
  - Define data-structures to represent programs
  - Define what is a program’s semantics (via an interpretation function)
  - Use parametric higher-order abstract syntax (PHOAS) to deal with binders

- Use Coq bindings to represent the bindings of the object language.

```
Section t.
 Variable var: T → Type.

 Inductive term : T → Type :=
 | Var: ∀ t, var t → term t
 | Abs: ∀ α β, (var α → term β) → term (α ⌜→⌝ β)
 | App: ...
End t.

Definition Term := ∀ (var: T → Type), term var.

Example K α β : Term (α ⌜→⌝ β ⌜→⌝ α):= fun V ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Var x)).
```

- An intrinsic approaches (strongly typed syntax vs. syntax + typing judgement)
- Program transformations are easier to implement (and prove!)

with one caveat

- Use Coq bindings to represent the bindings of the object language.

```
Section t.
 Variable var: T → Type.

 Inductive term : T → Type :=
 | Var: ∀ t, var t → term t
 | Abs: ∀ α β, (var α → term β) → term (α ⌜→⌝ β)
 | App: ...
End t.

Definition Term := ∀ (var: T → Type), term var.

Example K α β : Term (α ⌜→⌝ β ⌜→⌝ α):= fun V ⇒
  Abs (fun x ⇒ Abs (fun y ⇒ Var x)).
```

- An intrinsic approaches (strongly typed syntax vs. syntax + typing judgement)
- Program transformations are easier to implement (and prove!)

with one caveat

Outline

Fe-Si programs:

- update a set of memory elements Φ;

  registers, register files, fifos, ...

- are based on guarded atomic actions

  $$\mathtt{do}\ \mathtt{n} \leftarrow\ !\mathtt{x} + 1;\ (\mathtt{y} := 1;\ \mathtt{assert}\ (\mathtt{n} = 0))\ \mathtt{orElse}\ (\mathtt{y} := 2)$$

- are endowed with a (simple) synchronous semantics

  $$\mathtt{do}\ \mathtt{n} \leftarrow\ !\mathtt{x};\ \mathtt{x} := \mathtt{n} + 1;\ \mathtt{do}\ \mathtt{m} \leftarrow\ !\mathtt{x};\ \mathtt{assert}\ (\mathtt{n} = \mathtt{m})$$

Fe-Si in a nutshell

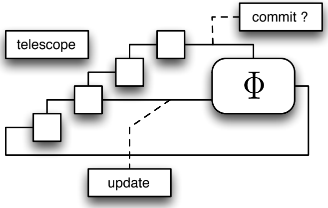
```
Variable V: T → Type.
Inductive 𝔸: T → Type:=
| Return: ∀ t, expr t → 𝔸 t
| Bind: ∀ t u, 𝔸 t → (V t → 𝔸 u) → 𝔸 u

(** effects **)
| Primitive: ...

(** control-flow **)
| OrElse: ∀ t, 𝔸 t → 𝔸 t → 𝔸 t.
| Assert: expr Tbool → 𝔸 Tunit
```

- Expressions are side-effects free.
- Primitives are operations on memory elements (dependent on Φ)
- `Definition Eval Φ t (a: ∀ V, 𝔸 V t): ⟦Φ⟧ → option (⟦t⟧ * ⟦Φ⟧).`

RTL in a nutshell




```
Variable V: T → Type.

Inductive 𝕋 (A: Type): Type:=
| Bind: ∀ arg, expr arg → (V arg → 𝕋 A) → 𝕋 A
| End: A → 𝕋 A.

Inductive 𝔼: memory → Type:=
| write: ∀ t, V t → V Tbool → 𝔼 (R t)
| ...

Definition block t:=
 𝕋 (V Tbool ∗ V t ∗ DList.T (option ∘ 𝔼) Φ).
```

- Simple synchronous semantics

```
Definition Eval Φ t (a: ∀ V, block V t): ⟦Φ⟧ → option (⟦t⟧ ∗ ⟦Φ⟧).
```

Running example:

```
do x ← ! r1;if (x <> 0) then {do y ← !r2; r1 := x − 1; r2 := y + 1} else { y ← !r2; r1 := y}
```

1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. WIP: Boolean simplification

Running example:

```
do x ← ! r1;if (x <> 0) then {do y ← !r2; r1 := x − 1; r2 := y + 1} else { y ← !r2; r1 := y}
```

1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. WIP: Boolean simplification

```
x0 ←  ! r1;
x1 ← x0 ≠ 0;
x2 ←  !r2;
x3 ← x0 − 1;
x4 ← x2 + 1;
x5 ←  !r2;
x6 ← x6;
begin
 if x1 then (r1 := x3; r2 := x4);
 if !x1 then (r1 := x6)
end
```

Running example:

$\texttt{do}\ x \leftarrow\ !\,r1; \texttt{if}\ (x <> 0)\ \texttt{then}\ \{\texttt{do}\ y \leftarrow\ !r2; r1 := x - 1; r2 := y + 1\}\ \texttt{else}\ \{ y \leftarrow\ !r2; r1 := y\}$

1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. WIP: Boolean simplification

```
x0  ← !r1;
x1  ← x0 ≠ 0;
x2  ← !r2;
x3  ← x0 − 1;
x4  ← x2 + 1;
x5  ← !r2;
x6  ← x5;
x8  ← x1;
x9 ← x1;
x10 ← not x1;
x11 ← x8 || x10;
x12 ← x8 ? x3 : x6;
begin
 r1 := x12 when x11;
 r2 := x4 when x9
end
```

Running example:

`do x ← ! r1;if (x <> 0) then {do y ← !r2; r1 := x − 1; r2 := y + 1} else { y ← !r2; r1 := y}`

1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. WIP: Boolean simplification

```
x0  ← !r1
x1  ← 0;
x2  ← x0 = x1;
x3  ← not x2;
x4  ← !r2;
x5  ← 1;
x6  ← x0 − x5;
x7  ← x4 + x5;
x8  ← !r2;
x9  ← not x3;
x10 ← x3 || x9
x11 ← x3 ? x6 : x8
begin
 r1 := x11 when x10;
 r2 := x8 when x3
end
```

Running example:

```
do x ← ! r1;if (x <> 0) then {do y ← !r2; r1 := x − 1; r2 := y + 1} else { y ← !r2; r1 := y}
```

1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. WIP: Boolean simplification

```
x0  ← !r1
x1  ← 0;
x2  ← x0 = x1;
x3  ← not x2;
x4  ← !r2;
x5  ← 1;
x6  ← x0 − x5;
x7  ← x4 + x5;
x8 ← x3 ? x6 : x4
begin
 r1 := x8 when true;
 r2 := x4 when x3
end
```

- (Temporary) final result

```
Definition Compile Φ t  (a : ∀ V, 𝔸 Φ V t) : ∀ V, block V Φ t :=
 let x := Flat.Compile Φ t (Push.Compile Φ  t (Pull.Compile Φ  t a)) in
 CSE.Compile Φ t x.

Theorem Compile_correct Φ t a :
 let x := Flat.Compile Φ t (Push.Compile Φ  t (Pull.Compile Φ  t a)) in
 WF Φ t x → ∀ (st : ⟦Φ⟧),  Eval Φ t (CSE.Compile Φ t x) st = Eval Φ t a st.
```

- No need to prove lemmas about substitutions
- What about `WF Φ t x` ?

PHOAS in action

- (Temporary) final result

```
Definition Compile Φ t  (a : ∀ V, 𝔸 Φ V t) : ∀ V, block V Φ t :=
 let x := Flat.Compile Φ t (Push.Compile Φ  t (Pull.Compile Φ  t a)) in
 CSE.Compile Φ t x.

Theorem Compile_correct Φ t a :
 let x := Flat.Compile Φ t (Push.Compile Φ  t (Pull.Compile Φ  t a)) in
 WF Φ t x → ∀ (st : ⟦Φ⟧),  Eval Φ t (CSE.Compile Φ t x) st = Eval Φ t a st.
```

- No need to prove lemmas about substitutions
- What about `WF Φ t x` ?

- (Temporary) final result

```
Definition Compile Φ t (a : ∀ V, 𝔸 Φ V t) : ∀ V, block V Φ t :=
 let x := Flat.Compile Φ t (Push.Compile Φ t (Pull.Compile Φ t a)) in
 CSE.Compile Φ t x.

Theorem Compile_correct Φ t a :
 let x := Flat.Compile Φ t (Push.Compile Φ t (Pull.Compile Φ t a)) in
 WF Φ t x → ∀ (st : ⟦Φ⟧), Eval Φ t (CSE.Compile Φ t x) st = Eval Φ t a st.
```

- No need to prove lemmas about substitutions
- What about `WF Φ t x` ?

Well-formedness

- `WF` Φ `t x` states that `x` is parametric w.r.t. the instantiation of `V`.
- We may:
  - posit $\forall$ `x`, `WF` Φ `t x` as an axiom (informed parties think that this is consistent with Coq)
  - or define what is `WF` for each language, prove that compilation preserves `WF` and prove that each starting program is `WF`
  - or generates `WF` Φ `t x` as a proof-obligation, and discharge it using tactics

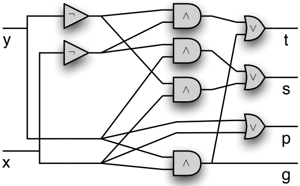
PHOAS in action (2)

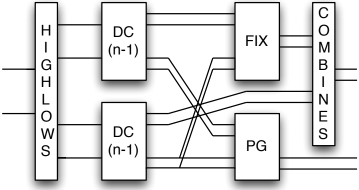
- PHOAS shines when defining examples of circuits inside Coq:
  - makes it possible to use fancy coq notations

    ```
    Notation "'DO' X ← A ; B" := (Bind A (fun X ⇒ B)) ( ... )
    ```

  - other solutions (e.g., dependently typed de Bruijn indices) would not scale
  - keep all the benefits of deep-embeddings!

Outline






```
Variable V : T → Type.

Fixpoint add Φ n (x : V (Tint [2^ n])) (y : V (Tint [2^ n])) :=
match n with
| 0 ⇒ Return ( (x = 1) ∨ (y = 1) ;
      (x = 1) ∧ (y = 1); x + y; x + y + 1)
| S n ⇒
  DO (xL,xH) ← (low x, high x);
  DO (yL,yH) ← (low y, high y);
  DO (pL, gL, sL, tL) ← add n xL yL;
  DO (pH, gH, sH, tH) ← add n xH yH;
  DO sH' ← (gL ? tH : sH);
  DO tH' ← (pL ? tH : sH);
  DO pH' ← (gH ∨ (pH ∧ gH));
  DO gH' ← (gH ∨ (pH ∧ gL));
  Return (pH'; gH'; sL ⊗ sH' ; tL ⊗ tH' )
end.
```

builds a 4-uple: carry-propagate, carry-generate, sum w/ carry, sum w/o carry

- Easy translation from old Bluespec papers

```
Definition bz :=
DO pc ← ! PC
DO I  ← IMEM.[pc] ;
WHEN (opcode I =  3 );
DO r1 ← RF.[r1 I];
DO r2 ← RF.[r2 I];
If  r1 = 0 { PC := r2 }
Else {PC := pc + 1}
```

```
(** Rule BZ taken **)
Proc(PC,RF,IMEM,DMEM)
if (RF[r1] = 0) where BZ(r1,r2) = IMEM[PC]
→ Proc(RF[r2],RF,IMEM,DMEM)

(** Rule BZ not taken **)
Proc(PC,RF,IMEM,DMEM)
if (RF[r1] <> 0) where BZ(r1,r2) = IMEM[PC]
→ Proc(PC + 1,RF,IMEM,DMEM)
```

- `Definition isa := loadi ⊕ loadpc ⊕  add ⊕ bz ⊕ load ⊕ store`
- Use a mixture of notations and intermediate definitions
- (Not yet tried to prove anything about this one)

Outline

Some remarks

- Stepping back
  - Bluespec started as an HDL deeply embedded in Haskell
  - Lava [1998] is another HDL deeply embedded in Haskell
  - Fe-Si is “just” another HDL, deeply embedded in Coq
    - semantics (i.e., interpreter), compiler and programs are integrated seamlessly
    - use of computation inside Coq to dump compiled programs
    - dependent types capture some interesting properties in hardware
- Future work
  - Improve on the language (inputs, FIFOs, schedulers)
  - Better compiler (boolean optimisations/BDDs)
  - Extraction/plugin to output actual VHDL/Verilog
  - Prove some designs correct (w.r.t. specifications as Moore automata)
- Closing remarks
  - Could really use some help from SMT solvers to solve bitvector arithmetic goals
  - Generated induction principles useless
  - Mutual fixpoints and inner fixpoints being not equivalent

Some remarks

- Stepping back
  - Bluespec started as an HDL deeply embedded in Haskell
  - Lava [1998] is another HDL deeply embedded in Haskell
  - Fe-Si is “just” another HDL, deeply embedded in Coq
    - semantics (i.e., interpreter), compiler and programs are integrated seamlessly
    - use of computation inside Coq to dump compiled programs
    - dependent types capture some interesting properties in hardware
- Future work
  - Improve on the language (inputs, FIFOs, schedulers)
  - Better compiler (boolean optimisations/BDDs)
  - Extraction/plugin to output actual VHDL/Verilog
  - Prove some designs correct (w.r.t. specifications as Moore automata)
- Closing remarks
  - Could really use some help from SMT solvers to solve bitvector arithmetic goals
  - Generated induction principles useless
  - Mutual fixpoints and inner fixpoints being not equivalent

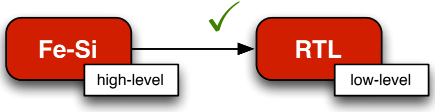
Some remarks

- Stepping back
  - Bluespec started as an HDL deeply embedded in Haskell
  - Lava [1998] is another HDL deeply embedded in Haskell
  - Fe-Si is “just” another HDL, deeply embedded in Coq
    - semantics (i.e., interpreter), compiler and programs are integrated seamlessly
    - use of computation inside Coq to dump compiled programs
    - dependent types capture some interesting properties in hardware
- Future work
  - Improve on the language (inputs, FIFOs, schedulers)
  - Better compiler (boolean optimisations/BDDs)
  - Extraction/plugin to output actual VHDL/Verilog
  - Prove some designs correct (w.r.t. specifications as Moore automata)
- Closing remarks
  - Could really use some help from SMT solvers to solve bitvector arithmetic goals
  - Generated induction principles useless
  - Mutual fixpoints and inner fixpoints being not equivalent

Some remarks

- Stepping back
  - Bluespec started as an HDL deeply embedded in Haskell
  - Lava [1998] is another HDL deeply embedded in Haskell
  - Fe-Si is "just" another HDL, deeply embedded in Coq
    - semantics (i.e., interpreter), compiler and programs are integrated seamlessly
    - use of computation inside Coq to dump compiled programs
    - dependent types capture some interesting properties in hardware
- Future work
  - Improve on the language (inputs, FIFOs, schedulers)
  - Better compiler (boolean optimisations/BDDs)
  - Extraction/plugin to output actual VHDL/Verilog
  - Prove some designs correct (w.r.t. specifications as Moore automata)
- Closing remarks
  - Could really use some help from SMT solvers to solve bitvector arithmetic goals
  - Generated induction principles useless
  - Mutual fixpoints and inner fixpoints being not equivalent

Thank you for your attention



If you have any questions ...