Formal verification of hardware synthesis

Thomas Braibant[1] Adam Chlipala[2]

Inria[1] (Gallium) MIT CSAIL[2]

SPADES's seminar

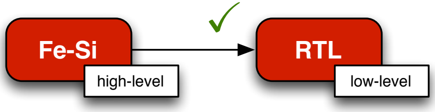
- Verifying hardware with theorem provers:
  - many formalizations of hardware description languages (ACL2 , HOL, PVS)
  - many models of hardware designs (ACL2, HOL, PVS, Coq)
    - Floating-point operations verified at AMD using ACL2
    - VAMP [2003] (a pipelined micro-processor verified in PVS)
  - high-level formalization of the ARM architecture in HOL
  - ...
- Shift toward hardware synthesis:
  - generates low-level code (RTL) from high-level HDLs
  - argue (in)formally that this synthesis is correct

Esterel, Lustre, System-C, Bluespec, ...

- Verifying hardware with theorem provers:
  - many formalizations of hardware description languages (ACL2 , HOL, PVS)
  - many models of hardware designs (ACL2, HOL, PVS, Coq)
    - Floating-point operations verified at AMD using ACL2
    - VAMP [2003] (a pipelined micro-processor verified in PVS)
  - high-level formalization of the ARM architecture in HOL
  - ...
- Shift toward hardware synthesis:
  - generates low-level code (RTL) from high-level HDLs
  - argue (in)formally that this synthesis is correct

Esterel, Lustre, System-C, Bluespec, ...

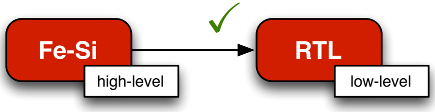
- Investigate verified hardware synthesis in Coq



- 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, single-clock, unit delays

This project

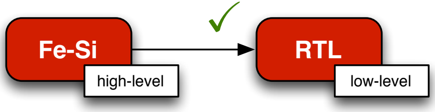
- Investigate verified hardware synthesis in Coq



- 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, single-clock, unit delays

This project

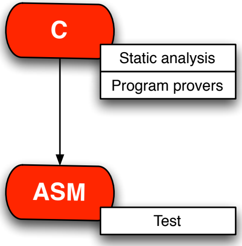
- Investigate verified hardware synthesis in Coq

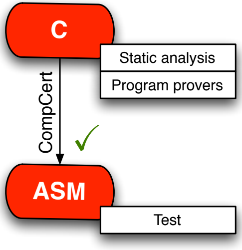


- Source language: Fe-Si (Featherweight Synthesis)
  - Stripped down and simplified version of Bluespec
  - Semantics based on "guarded atomic actions" (with a flavour of transactional memory)
- Target language: RTL
  - Combinational logic and next-state assignments for registers
  - No currents, single-clock, unit delays

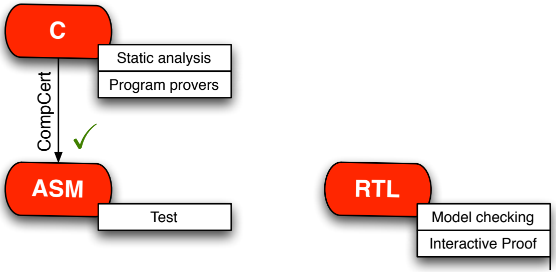
Outline

Synthesis from the perspective of verification

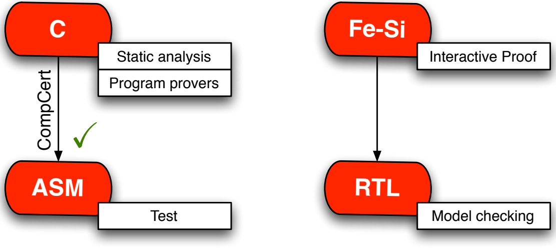



C
Static analysis
Program provers
CompCert
ASM
Test

Synthesis from the perspective of verification



Synthesis from the perspective of verification



- Define deep-embeddings
  - Define data-structures to represent programs (a prerequisite to write a compiler)
  - Define what is a program's semantics
- Implement the compiler
- Pick a phrasing for semantic preservation:

$$\mathcal{B}(P_1) \,\square\, \mathcal{B}(P_2) \qquad \begin{cases} \square \in \{\subseteq, \supseteq, \equiv\} \\ \text{deterministic } P_2? \\ \text{safe } P_1? \end{cases}$$

- Prove semantic preservation for your compiler.

easier said than done

A certified compiler in a nutshell

- Define deep-embeddings
  - Define data-structures to represent programs (a prerequisite to write a compiler)
  - Define what is a program's semantics
- Implement the compiler
- Pick a phrasing for semantic preservation:

$$\mathcal{B}(P_1) \,\square\, \mathcal{B}(P_2) \qquad \begin{cases} \square \in \{\subseteq, \supseteq, \equiv\} \\ \text{deterministic } P_2? \\ \text{safe } P_1? \end{cases}$$

- Prove semantic preservation for your compiler.

easier said than done

- Define deep-embeddings
  - Define data-structures to represent programs (a prerequisite to write a compiler)
  - Define what is a program's semantics
- Implement the compiler
- Pick a phrasing for semantic preservation:

$$\mathcal{B}(P_1) \square \mathcal{B}(P_2) \qquad \begin{cases} \square \in \{\subseteq, \supseteq, \equiv\} \\ \text{deterministic } P_2? \\ \text{safe } P_1? \end{cases}$$

- Prove semantic preservation for your compiler.

easier said than done

A certified compiler in a nutshell

- Define deep-embeddings
  - Define data-structures to represent programs (a prerequisite to write a compiler)
  - Define what is a program's semantics
- Implement the compiler
- Pick a phrasing for semantic preservation:

$$\mathcal{B}(P_1) \square \mathcal{B}(P_2) \qquad \begin{cases} \square \in \{\subseteq, \supseteq, \equiv\} \\ \text{deterministic } P_2\text{?} \\ \text{safe } P_1\text{?} \end{cases}$$

- Prove semantic preservation for your compiler.

easier said than done

- Define deep-embeddings
  - Define data-structures to represent programs (a prerequisite to write a compiler)
  - Define what is a program's semantics
- Implement the compiler
- Pick a phrasing for semantic preservation:

$$\mathcal{B}(P_1) \square \mathcal{B}(P_2) \qquad \begin{cases} \square \in \{\subseteq, \supseteq, \equiv\} \\ \text{deterministic } P_2? \\ \text{safe } P_1? \end{cases}$$

- Prove semantic preservation for your compiler.

easier said than done

Extra goals:

- make it easy to write source programs inside Coq;
- make it relatively easy to reason about them.

Problem: abstract syntax

```
let x = foo in
let y = f x in
let z = g x y in
z + x
```

```
let foo in
let f #1 in
let g #2 #1 in
#1 + #3
```

- A complicated problem, many solutions, no clear winner;
- Here, hijack Coq binders using Parametric Higher-Order Abstract Syntax (PHOAS)

Extra goals:

- make it easy to write source programs inside Coq;
- make it relatively easy to reason about them.

Problem: abstract syntax

```
let x = foo in
let y = f x in
let z = g x y in
z + x
```

```
let foo in
let f #1 in
let g #2 #1 in
#1 + #3
```

- A complicated problem, many solutions, no clear winner;
- Here, hijack Coq binders using Parametric Higher-Order Abstract Syntax (PHOAS)

- 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 approach (strongly typed syntax vs. syntax + typing judgement)

- 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 approach (strongly typed syntax vs. syntax + typing judgement)

To sum up

- High-level languages have more structure.

  easier for verification.

- Certified compilers are semantics preserving.

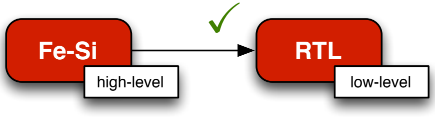
  transport verification to low-level languages.

- Extra difficulty in our case: nested binders.

  solved using PHOAS.

Outline

The big picture



- Based on a monad
- Base constructs: bind and return

```
Definition hadd (a b: Var B) : action [] (B ⊗ B) :=
  do carry ← ret (andb a b);
  do sum   ← ret (xorb a b);
  ret (carry, sum).
```

- Set of memory elements to hold mutable state

```
Definition count n : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← member_0 ::= x + 1;
  ret x.
```

- Control-flow constructions

```
Definition count n (tick: Var B) : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← if tick then {member_0 ::= x + 1} else {ret ()};
  ret x.
```

Fe-Si, informally

- Based on a monad
- Base constructs: bind and return

```
Definition hadd (a b: Var B) : action [] (B ⊗ B) :=
  do carry ← ret (andb a b);
  do sum   ← ret (xorb a b);
  ret (carry, sum).
```

- Set of memory elements to hold mutable state

```
Definition count n : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← member_0 ::= x + 1;
  ret x.
```

- Control-flow constructions

```
Definition count n (tick: Var B) : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← if tick then {member_0 ::= x + 1} else {ret ()};
  ret x.
```

- Based on a monad
- Base constructs: bind and return

```
Definition hadd (a b: Var B) : action [] (B ⊗ B) :=
  do carry ← ret (andb a b);
  do sum   ← ret (xorb a b);
  ret (carry, sum).
```

- Set of memory elements to hold mutable state

```
Definition count n : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← member_0 ::= x + 1;
  ret x.
```

- Control-flow constructions

```
Definition count n (tick: Var B) : action [Reg (Int n)] (Int n) :=
  do x ← !member_0;
  do _ ← if tick then {member_0 ::= x + 1} else {ret ()};
  ret x.
```

Fe-Si programs:

- update a set of memory elements Φ;

  registers, register files, inputs, …

- are based on guarded atomic actions

  `do n ← !x + 1; (y := 1; assert (n = 0)) orElse (y := 2)`

- are endowed with a (simple) synchronous semantics

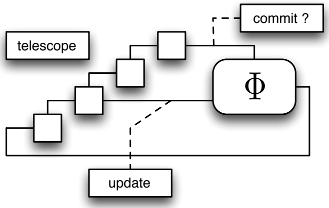
  `do n ← !x; x := n + 1; do m ← !x; assert (n = m)`

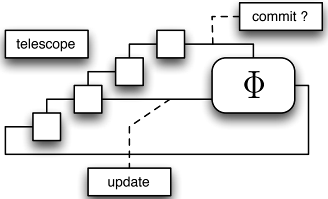
```
Variable var: ty → Type.
Inductive expr: ty → Type := ...

Inductive action: ty → Type:=
| Return: ∀ t, expr t → action t
| Bind: ∀ t u, action t → (var t → action u) → action u
(** control-flow **)
| OrElse: ∀ t, action t → action t → action t
| Assert: expr B → action unit
(** memory operations on registers **)
| Read: ∀ t, (Reg t) ∈ Φ → action t
| Writ: ∀ t, (Reg t) ∈ Φ → expr t → action unit
| ...
```

- Expressions are side-effects free.
- `Definition Eval Φ t (a: ∀ V, action V t): ⟦Φ⟧ → option (⟦t⟧ * ⟦Φ⟧).`

An RTL circuit is abstracted as:

- a set of memory elements Φ;
- a combinational next-state function.






```
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. 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. 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:

```
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. 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. 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
```

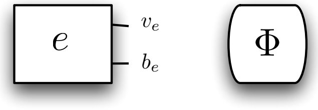
Running example:

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

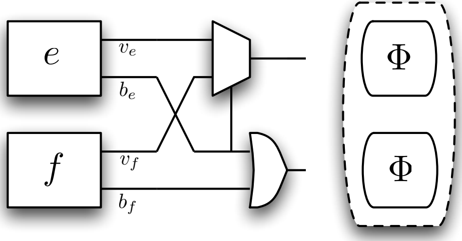
1. Pull out all bindings (that is, ANF)
2. Push down the nested conditions
3. Perform CSE (in 3-address code)
4. Boolean simplification

```
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
```

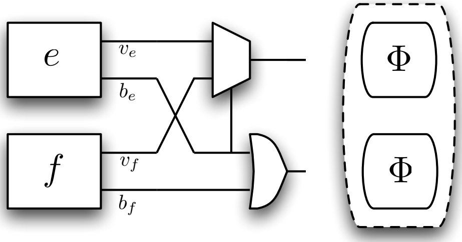
- Transform control-flow into data-flow.



- Transform control-flow into data-flow.
- Compiling `e orElse f`



- Transform control-flow into data-flow.
- Compiling `e orElse f`



- The Φ blocks are trees of effects on memory elements, that must be flattened.

1. Transform control-flow into data-flow programs (in A-normal form);
2. Compute the update and commit values for each memory element;
3. Perform syntactic common-sub-expression elimination;
4. Perform Boolean expressions reduction using BDDs;
5. Use an OCaml backend to generate Verilog code.

- Steps [1-4] are proved correct in Coq.

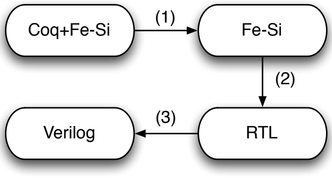
Not a single lemma about substitutions!

1. Transform control-flow into data-flow programs (in A-normal form);
2. Compute the update and commit values for each memory element;
3. Perform syntactic common-sub-expression elimination;
4. Perform Boolean expressions reduction using BDDs;
5. Use an OCaml backend to generate Verilog code.

- Steps [1-4] are proved correct in Coq.

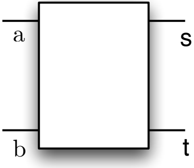
Not a single lemma about substitutions!

Outline



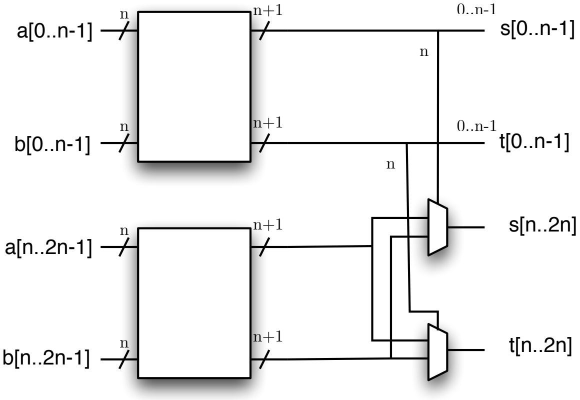
1. Coq's reduction (internal)
2. Fe-Si to RTL (proved)
3. RTL to Verilog (OCaml backend, trusted)

First example: A recursive von Neumann adder



$$s = a + b \quad t = a + b + 1$$

First example: A recursive von Neumann adder



$$s = a + b \quad t = a + b + 1$$

```
Variable V : T → Type.

Fixpoint add Φ n (a : V (Tint [2^ n])) (b : V (Tint [2^ n])) :=
match n  with
| 0 ⇒ ret ( (a = 1) ∨ (b = 1) ;
            (a = 1) ∧ (b = 1); a + b; a + b + 1)
| S n ⇒
  do (aL,aH) ← (low a, high a);
  do (bL,bH) ← (low b, high b);
  do (pL, gL, sL, tL) ← add n aL bL;
  do (pH, gH, sH, tH) ← add n aH bH;
  do sH′ ← (gL ? tH : sH);
  do tH′ ← (pL ? tH : sH);
  do pH′ ← (gH ∨ (pH ∧ gH));
  do gH′ ← (gH ∨ (pH ∧ gL));
  ret (pH′; gH′; sL ⊗ sH′ ; tL ⊗ tH′ )
end.
```

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

- Proof by induction on *n*

First example: A recursive von Neumann adder

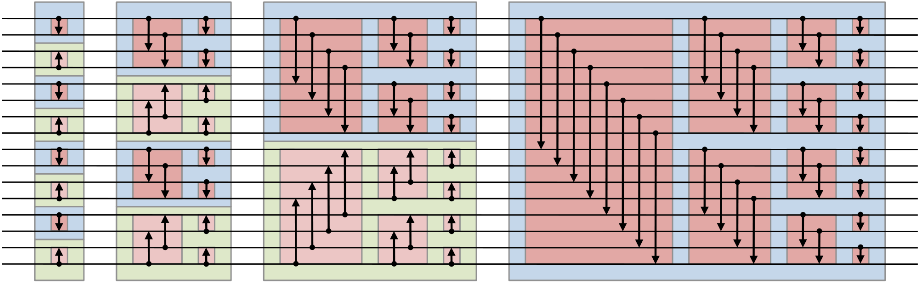
Meta-programming for free

```
Variable V : T → Type.

Fixpoint add Φ n (a : V (Tint [2^ n])) (b : V (Tint [2^ n])) :=
match n  with
| 0 ⇒ ret ( (a = 1) ∨ (b = 1) ;
            (a = 1) ∧ (b = 1); a + b; a + b + 1)
| S n ⇒
  do (aL,aH) ← (low a, high a);
  do (bL,bH) ← (low b, high b);
  do (pL, gL, sL, tL) ← add n aL bL;
  do (pH, gH, sH, tH) ← add n aH bH;
  do sH′ ← (gL ? tH : sH);
  do tH′ ← (pL ? tH : sH);
  do pH′ ← (gH ∨ (pH ∧ gH));
  do gH′ ← (gH ∨ (pH ∧ gL));
  ret (pH′; gH′; sL ⊗ sH′ ; tL ⊗ tH′ )
end.
```

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

- Proof by induction on n

Second example: a bitonic sorter core



- Bitonic: $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ for some k, or a circular shift.
- Red: bitonic $\rightarrow l_1$ bitonic, l_2 bitonic (and $l_1 \leq l_2$)
- Blue (resp. green): bitonic $\rightarrow$ sorted (resp. sorted in reverse order)

Second example: a bitonic sorter code

```
Fixpoint merge {n}: T n → C n :=
match n with
| 0 ⇒ fun t ⇒ ret (leaf t)
| S k ⇒ fun t ⇒
  do a,b ⇜ min_max_swap (left t) (right t);
  do a ← merge a;
  do b ← merge b;
  ret (mk_N a b)
end.
```

```
Fixpoint sort {n} : T n → C n :=
match n with
| 0 ⇒ fun t ⇒ ret (leaf t)
| S n ⇒ fun t ⇒
  do l ← sort (left t);
  do r ⇜ sort (right t);
  do r ← reverse r;
  do x ⇜ ret (mk_N l r);
  merge x
end
```

- merge builds the blue/green boxes
- min_max_swap builds the red boxes
- Notations

```
Notation T n := tree (expr Var A) n.
Notation C n := action nil Var (domain n).
```

- Step-stone: implement a purely functional version of the sorting algorithm;
- Prove that the sorter core and the step-stone function have the same semantics;
- Prove that the step-stone is correct.

Theorem

Let I be a sequence of length 2^n of integers of size m. The circuit always produces an output sequence that is a sorted permutation of I.

- Step-stone: implement a purely functional version of the sorting algorithm;
- Prove that the sorter core and the step-stone function have the same semantics;
- Prove that the step-stone is correct.

Theorem (0-1 principle)

To prove that a (parametric) sorting network is correct, it suffices to prove that it sorts all sequences of 0 and 1.

Theorem

Let l be a sequence of length 2^n of integers of size m. The circuit always produces an output sequence that is a sorted permutation of l.

- Step-stone: implement a purely functional version of the sorting algorithm;
- Prove that the sorter core and the step-stone function have the same semantics;
- Prove that the step-stone is correct.

Theorem

Let I be a sequence of length 2^n of integers of size m. The circuit always produces an output sequence that is a sorted permutation of I.

$$
\begin{array}{rclll}
\mathrm{i} & ::= & \texttt{const}\ n & \vdash pc, \sigma, s \rightarrow pc+1, n :: \sigma, s & \\
& | & \texttt{var}\ x & \vdash pc, \sigma, s \rightarrow pc+1, s(x) :: \sigma, s & \\
& | & \texttt{setvar}\ x & \vdash pc, v :: \sigma, s \rightarrow pc+1, \sigma, s[x \leftarrow v] & \\
& | & \texttt{add} & \vdash pc, n_2 :: n_1 :: \sigma, s \rightarrow pc+1, (n_1+n_2) :: \sigma, s & \\
& | & \texttt{sub} & \vdash pc, n_2 :: n_1 :: \sigma, s \rightarrow pc+1, (n_1-n_2) :: \sigma, s & \\
& | & \texttt{bfwd}\ \delta & \vdash pc, \sigma, s \rightarrow pc+1+\delta, \sigma, s & \\
& | & \texttt{bbwd}\ \delta & \vdash pc, \sigma, s \rightarrow pc+1-\delta, \sigma, s & \\
& | & \texttt{bcond}\ c\ \delta & \vdash pc, n_2 :: n_1 :: \sigma, s \rightarrow pc+1+\delta, \sigma, s & \text{if } c\ n_1\ n_2 \\
& & & \vdash pc, n_2 :: n_1 :: \sigma, s \rightarrow pc+1, \sigma, s & \text{if } \neg(c\ n_1\ n_2) \\
& | & \texttt{halt} & \texttt{no reduction} &
\end{array}
$$

- Implementation parameterized by the size of the values, the size of the stack, ...

```
Definition pop :=
do sp ← ! SP;
do v ← read STACK [: sp − 1];
do _ ← SP ::= sp − 1;
ret v.
```

```
Definition Isetvar pc x :=
do v ← pop;
do _ ← write REGS [: x ← v];
PC ::= pc + 1.
```

$$\vdash pc, v :: \sigma, s \rightarrow pc + 1, \sigma, s[x \leftarrow v]$$

- Combine these pieces of code using a `case` construct
- Prove the Fe-Si implementation correct w.r.t. the previous semantics
- Fun fact: can run binary blobs generated by e.g., an IMP compiler

(an highly stylised way to compute Fibonacci)

- Recursive circuits.
- Combinators and schedulers of atomic actions.
- In these cases, using Coq as a meta-language makes it possible to prove things.

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
    - dependent types capture some interesting properties in hardware
    - use of extraction to dump compiled programs
- Take-away message
  - Coq can be used as an embedding language for DSLs!

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
    - dependent types capture some interesting properties in hardware
    - use of extraction to dump compiled programs
- Take-away message
  - Coq can be used as an embedding language for DSLs!

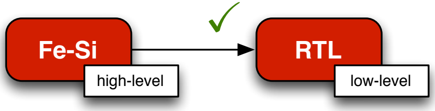
Some remarks

- 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
    - dependent types capture some interesting properties in hardware
    - use of extraction to dump compiled programs
- Take-away message
  - Coq can be used as an embedding language for DSLs!

On going work

- Recent work
  - Three BDD libraries (with J.-H. Jourdan and D. Monniaux, ITP 2013): reflections on the implementation of hash-consing.
  - A better inversion tactic (with P. Boutillier)
- Future work
  - Improve on the language (FIFOs, references).
  - Make the compiler more realistic (automatic scheduling of atomic actions).
  - Embed a DSL for floating-point cores?
  - More control!

Thank you for your attention



If you have any questions ...