Circuits in Coq: a tale of encodings

Thomas Braibant

Inria

GT Théorie des types et réalisabilité

Outline: Formalize and reason about circuits

- First encoding: graphical (categorical) approach
- Second encoding: scaling up, handling variables properly
- (if time permits): encodings of BDD libraries

(Bredth-first talk!)

Outline: Formalize and reason about circuits

- First encoding: graphical (categorical) approach
- Second encoding: scaling up, handling variables properly
- (if time permits): encodings of BDD libraries

(Bredth-first talk!)

Formalising circuits in proof assistants

(Circa 1985)

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:
- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Formalising circuits in proof assistants

(Circa 1985)

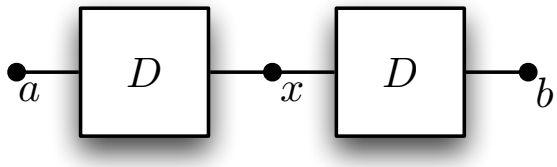
Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

$$D(a, x) \wedge D(x, b)$$



- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Formalising circuits in proof assistants

(Circa 1985)

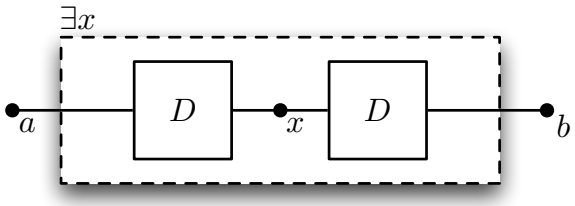
Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

$$\exists x, D(a, x) \wedge D(x, b)$$



- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Formalising circuits in proof assistants

(Circa 1985)

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

Composition

$$D(a, x) \wedge D(x, b)$$

Hiding

$$\exists x, D(a, x) \wedge D(x, b)$$

- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Formalising circuits in proof assistants

(Circa 1985)

Representing circuits with predicates (or functions).

- Some definitions:

$$Xor(i_1, i_2, o) \triangleq (o = \neg(i_1 = i_2)) \qquad Not(i, o) \triangleq (o = \neg i)$$

- Adding structure:

Composition

$$D(a, x) \wedge D(x, b)$$

Hiding

$$\exists x, D(a, x) \wedge D(x, b)$$

- Correctness proof: entailment of a specification.

$$(\exists x, Xor(i_1, i_2, x) \wedge Not(x, o)) \implies (o = (i_1 = i_2))$$

Shallow-embeddings vs deep-embeddings

Using a shallow-embedding, there is no way to:

- restrict the quantification on circuits;
- reason on the structure of the circuit in the proof assistant.

Move to deep-embeddings:

- define a data structure for circuits;
- define what's a circuit semantics (via an interpretation function);
- prove that a device implements a given specification.

No currents, unit delays

Shallow-embeddings vs deep-embeddings

Using a shallow-embedding, there is no way to:

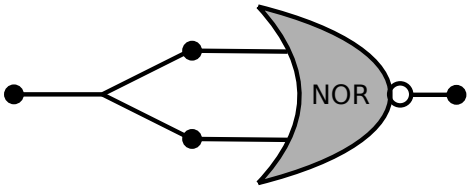
- restrict the quantification on circuits;
- reason on the structure of the circuit in the proof assistant.

Move to deep-embeddings:

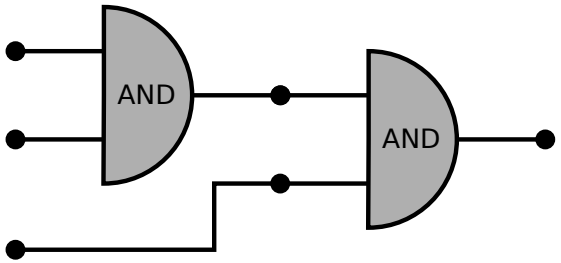
- define a data structure for circuits;
- define what's a circuit semantics (via an interpretation function);
- prove that a device implements a given specification.

No currents, unit delays

The graphical approach

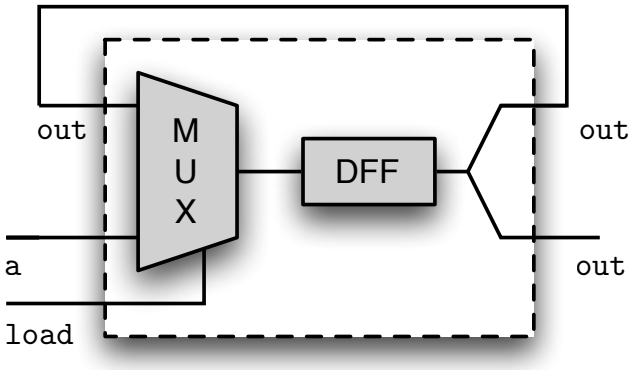



```
Fork 2 ▷ NOR
Not : circuit 1 1
```




```
(AND & One 1) ▷ AND
And3 : circuit 3 1
```

The graphical approach



A dependent type for circuits in Coq

- First version:

```
Inductive ℂ : nat → nat → Type := ...
```

- Examples:

$$\begin{array}{lcl} \texttt{Not} & : & \mathbb{C}\ 1\ 1 \\ \texttt{And3} & : & \mathbb{C}\ 3\ 1 \\ \texttt{Adder}\ n & : & \mathbb{C}\ (2n+1)\ (n+1) \end{array}$$

- Does not give much structure!

$$\begin{aligned} 2n+1 &= n+n+1 \\ &= n+1+n \\ &= 1+n+n \end{aligned}$$

A dependent type for circuits in Coq

- First version:

```
Inductive ℂ : nat → nat → Type := ...
```

- Examples:

$$\begin{array}{lcl} \texttt{Not} & : & \mathbb{C}\ 1\ 1 \\ \texttt{And3} & : & \mathbb{C}\ 3\ 1 \\ \texttt{Adder}\ n & : & \mathbb{C}\ (2n+1)\ (n+1) \end{array}$$

- Does not give much structure!

$$\begin{aligned} 2n+1 &= n+n+1 \\ &= n+1+n \\ &= 1+n+n \end{aligned}$$

A better dependent type for circuits in Coq

- Let'suse arbitrary types as indexes for the ports:

  `Inductive` $\mathbb{C}$: `Type` $\to$ `Type` $\to$ `Type` := ...

- For instance ($\mathbf{1}$ is the unit type, and $\oplus$ is disjoint-sum):

`Not`	:	$\mathbb{C}\ \mathbf{1}\ \mathbf{1}$	semantics$(\mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`And3`	:	$\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ \mathbf{1}$	semantics $(\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`Adder` n	:	$\mathbb{C}\ (n \cdot \mathbf{1} \oplus n \cdot \mathbf{1} \oplus \mathbf{1})\ (n \cdot \mathbf{1} \oplus \mathbf{1})$	...

Unsurprising: circuits are morphisms between sets of ports

A better dependent type for circuits in Coq

- Let'suse arbitrary types as indexes for the ports:

  `Inductive` $\mathbb{C}$: `Type` $\to$ `Type` $\to$ `Type` := ...

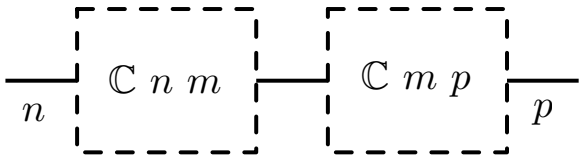
- For instance ($\mathbf{1}$ is the unit type, and $\oplus$ is disjoint-sum):

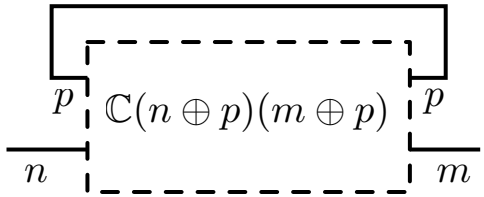
`Not`	:	$\mathbb{C}\ \mathbf{1}\ \mathbf{1}$	semantics$(\mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`And3`	:	$\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ \mathbf{1}$	semantics $(\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`Adder` n	:	$\mathbb{C}\ (n \cdot \mathbf{1} \oplus n \cdot \mathbf{1} \oplus \mathbf{1})\ (n \cdot \mathbf{1} \oplus \mathbf{1})$	...

Unsurprising: circuits are morphisms between sets of ports

Combinators

- Composition: $\mathbb{C}\ n\ m \to \mathbb{C}\ m\ p \to \mathbb{C}\ n\ p$
- Product: $\mathbb{C}\ n\ m \to \mathbb{C}\ p\ q \to \mathbb{C}\ (n \oplus p)\ (m \oplus q)$
- Trace: $\mathbb{C}\ (n \oplus p)\ (m \oplus p) \to \mathbb{C}\ n\ m$





Wiring

- Nameless setting: wires have to be forked, reordered!
- Could use symmetries and diagonals for wiring.

Wiring

- Nameless setting: wires have to be forked, reordered!
- Could use symmetries and diagonals for wiring.
- To tame complexity, use plugs
- A plug is a circuit of type $\mathbb{C}\ n\ m$ defined as a map from m to n

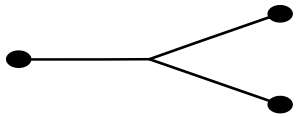
Wiring

- Nameless setting: wires have to be forked, reordered!
- Could use symmetries and diagonals for wiring.
- To tame complexity, use plugs
- A plug is a circuit of type $\mathbb{C}\ n\ m$ defined as a map from m to n
- Examples

(a)

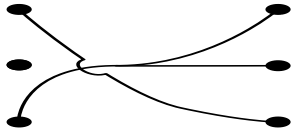
$\mathbb{C}\ (n \oplus m)\ m$

(b)



$\mathbb{C}\ n\ (n \oplus n)$

(c)



$\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

types must be read bottom-up

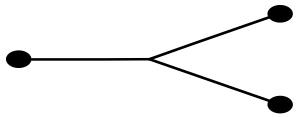
Wiring

- Nameless setting: wires have to be forked, reordered!
- Could use symmetries and diagonals for wiring.
- To tame complexity, use plugs
- A plug is a circuit of type $\mathbb{C}\ n\ m$ defined as a map from m to n
- Examples

(a)

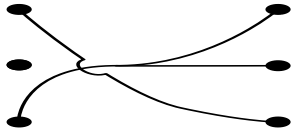
$\mathbb{C}\ (n \oplus m)\ m$

(b)



$\mathbb{C}\ n\ (n \oplus n)$

(c)



$\mathbb{C}\ (n \oplus m \oplus p)\ (p \oplus (n \oplus n))$

types must be read bottom-up

a) `fun (x : m) ⇒ inr n x`

b) `fun (x : n ⊕ n) ⇒ match x with inl e ⇒ e | inr e ⇒ e end.`

c)
```
fun (x : p ⊕ (n ⊕ n)) ⇒ match x with
| inl ep ⇒ inr (n ⊕ m) ep
| inr (inl en) ⇒ inl p (inl m en)
| inr (inr en) ⇒ inl p (inl m en)
```

by proof-search

So far, so good

- Strongly typed syntax (also, intrinsic approach)

```
Inductive ℂ : Type → Type → Type :=
| Atom : ∀ (n m : Type), atom n m → ℂ n m
| Plug : ∀ (n m : Type) (f : m → n), ℂ n m
| Ser  : ∀ (n m p : Type), ℂ n m → ℂ m p → ℂ n p
| Par  : ∀ (n m p q : Type), ℂ n p → ℂ m q →  ℂ (n ⊕ m) (p ⊕ q)
| Loop : ∀ (n m p : Type), ℂ (n ⊕ p) (m ⊕ p) → ℂ n m.
```

- Semantics of a circuit of type $\mathbb{C}\ n\ m$ induces a relation between $n \to \mathbb{T}$ and $m \to \mathbb{T}$.

```
Inductive sem: ∀ (n m: Type) (c: ℂ n m) → (n → 𝕋) → (m → 𝕋) → Prop := ...
```

Using isomorphisms

- The semantics relates valuations (e.g.: $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$).
- We need to reason up-to type isomorphisms
- Examples:

$$\frac{}{\mathbf{1} \to \mathbb{T} \cong \mathbb{T}} \qquad \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \frac{}{\mathbf{0} \to \mathbb{T} \cong \mathbf{1}}$$

- Use it to define specifications.

Using isomorphisms

- The semantics relates valuations (e.g.: $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$).
- We need to reason up-to type isomorphisms
- Examples:

$$\frac{}{\mathbf{1} \to \mathbb{T} \cong \mathbb{T}} \qquad \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \frac{}{\mathbf{0} \to \mathbb{T} \cong \mathbf{1}}$$

- Use it to define specifications.

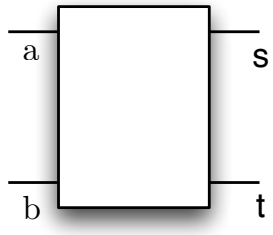
Using isomorphisms

- The semantics relates valuations (e.g.: $\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}$).
- We need to reason up-to type isomorphisms
- Examples:

$$\frac{}{\mathbf{1} \to \mathbb{T} \cong \mathbb{T}} \qquad \frac{A \to \mathbb{T} \cong \sigma \qquad B \to \mathbb{T} \cong \tau}{A \oplus B \to \mathbb{T} \cong (\sigma \times \tau)} \qquad \frac{}{\mathbf{0} \to \mathbb{T} \cong \mathbf{1}}$$

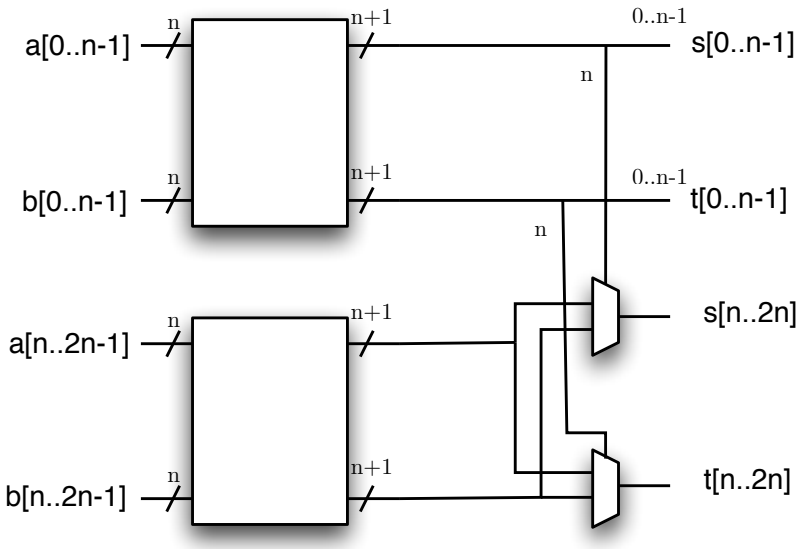
- Use it to define specifications.

An adder



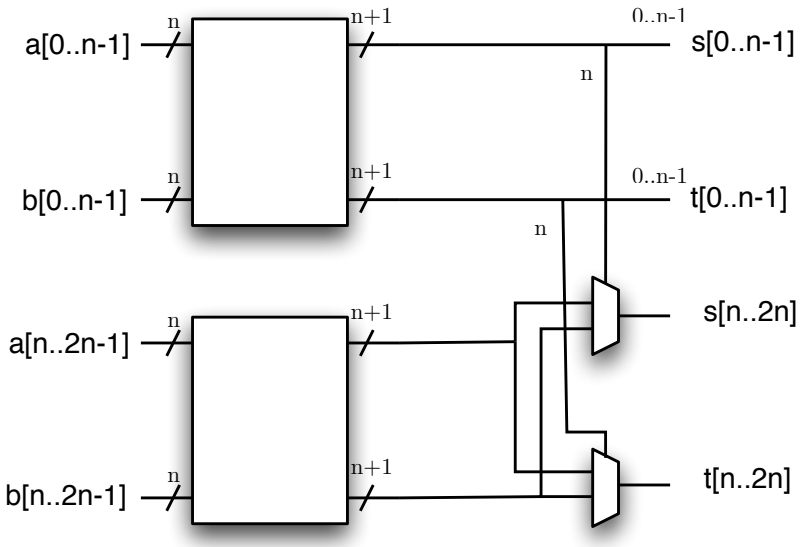
$$s = a + b \quad t = a + b + 1$$

An adder



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

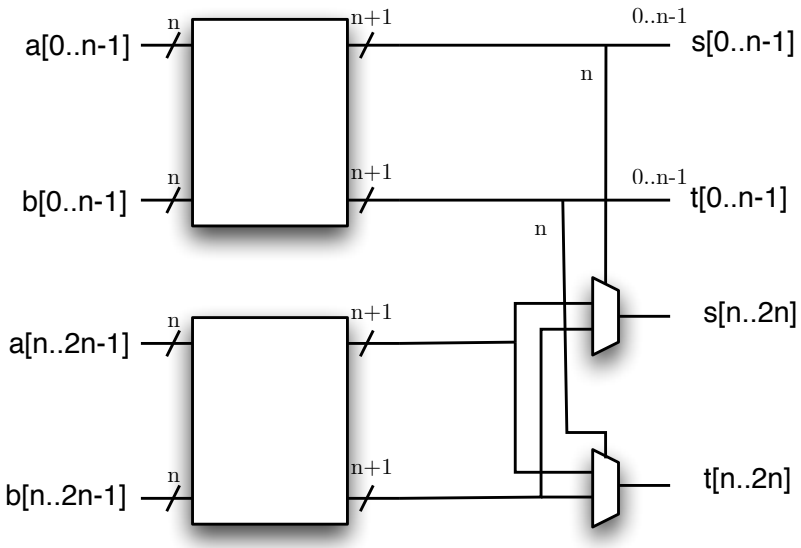
An adder



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

Up-to isomorphisms, this circuit implements the addition of bitvectors of size n

An adder



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

Problem: there is too much entanglement

- spend lots of time on wiring
- spend lots of time on basic stuff (reordering the context)

Diagramatic reasoning

- is appealing
- yet, is not suited for “real” verification efforts
- requires much boiler-plate

variables are just too important

Let's take binders seriously

Many ways to deal with nested binders

- Named variables
- (Dependently typed) De Bruijn indices;
- Locally nameless;
- ...

Let's take binders seriously

Many ways to deal with nested binders

- Named variables
- (Dependently typed) De Bruijn indices;
- Locally nameless;
- ...
- Parametric Higher-Order Abstract Syntax (PHOAS)

A PHOAS primer

- 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: ∀ α β, term (α ⌜→⌝ β) → term α → term β
End t.

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

- Example: K.

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

A PHOAS primer (cont.)

- First example: count the size of a term.

```
Fixpoint size t (e : term (fun _ ⇒ unit) t) : nat :=
match e with
| Var t v ⇒ 1
| Abs alpha beta e ⇒ 1 + size (e tt)
| App _ _ e f ⇒ 1+ size e + size f
end.
```

- Second example: compute the denotation of a term.

```
Fixpoint denote t (e : term ⟦·⟧ t) : ⟦t⟧ :=
match e with
| Var t v ⇒ v
| Abs alpha beta e ⇒ fun x ⇒ denote (e x)
| App _ _ e f ⇒ (denote e) (denote f)
end.
```

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

with one caveat

A PHOAS primer (cont.)

- First example: count the size of a term.

```
Fixpoint size t (e : term (fun _ ⇒ unit) t) : nat :=
match e with
| Var t v ⇒ 1
| Abs alpha beta e ⇒ 1 + size (e tt)
| App _ _ e f ⇒ 1+ size e + size f
end.
```

- Second example: compute the denotation of a term.

```
Fixpoint denote t (e : term ⟦·⟧ t) : ⟦t⟧ :=
match e with
| Var t v ⇒ v
| Abs alpha beta e ⇒ fun x ⇒ denote (e x)
| App _ _ e f ⇒ (denote e) (denote f)
end.
```

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

with one caveat

A PHOAS primer (cont.)

- First example: count the size of a term.

```
Fixpoint size t (e : term (fun _ ⇒ unit) t) : nat :=
match e with
| Var t v ⇒ 1
| Abs alpha beta e ⇒ 1 + size (e tt)
| App _ _ e f ⇒ 1+ size e + size f
end.
```

- Second example: compute the denotation of a term.

```
Fixpoint denote t (e : term ⟦·⟧ t) : ⟦t⟧ :=
match e with
| Var t v ⇒ v
| Abs alpha beta e ⇒ fun x ⇒ denote (e x)
| App _ _ e f ⇒ (denote e) (denote f)
end.
```

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

with one caveat

Back to circuit generators

with PHOAS

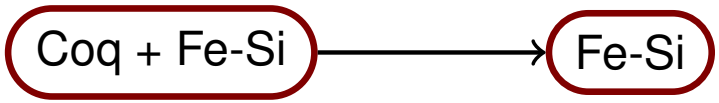
Fe-Si

Joint work with A. Chlipala

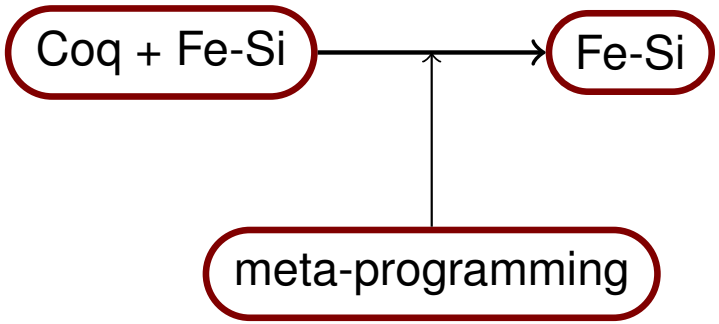
Fe-Si (Featherweigth synthesis)

- an Embedded Domain Specific Language in Coq
- an idealized subset of the Bluespec hardware description language (MIT, industrial)
- describe circuits in “term rewriting systems” style
- semantics based on guarded atomic actions
- a “surface” language for circuit generators;
- with a compiler to a more “primitive” one (RTL)

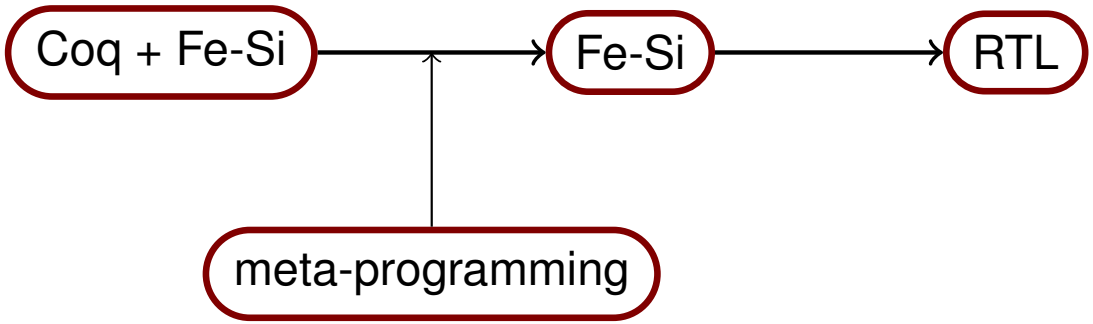
Why do we need both a compiler and meta-programming?



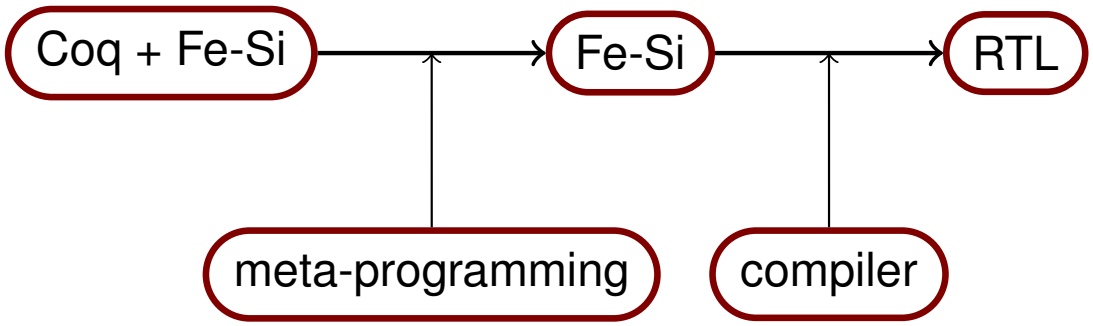
Why do we need both a compiler and meta-programming?



Why do we need both a compiler and meta-programming?

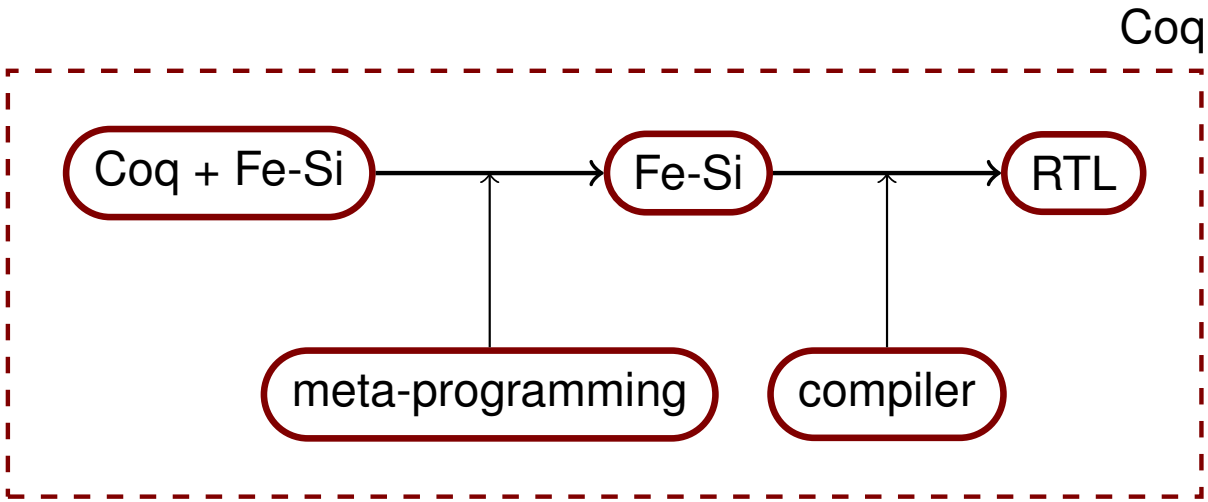


Why do we need both a compiler and meta-programming?



Our flavor of RTL is defined in terms of combinational logic and registers

Why do we need both a compiler and meta-programming?



Our flavor of RTL is defined in terms of combinational logic and registers

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.

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.
```

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.
```

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.
```

Fe-Si's semantics

Fe-Si programs:

- update a set of memory elements Φ;

registers, register files, inputs, ...

- are based on guarded atomic actions

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

- are endowed with a (simple) synchronous semantics

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

Fe-Si, formally

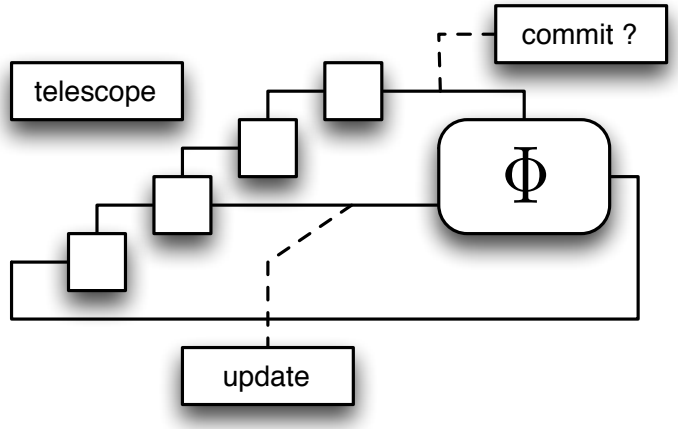
```
Variable var: ty → Type.
Inductive expr: ty → Type := ...

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⟧).`

RTL abreviated

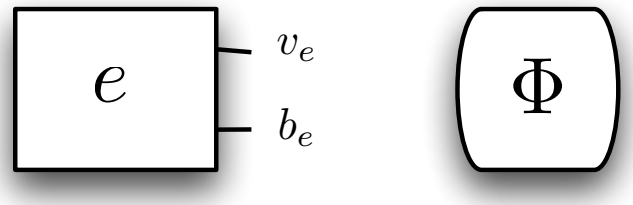
An RTL circuit is abstracted as:

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



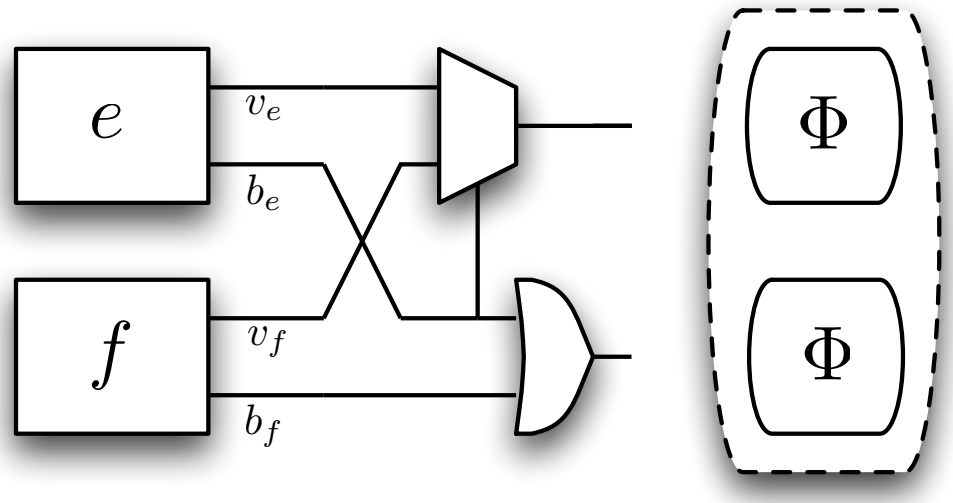
A glimpse of the first pass

- Transform control-flow into data-flow.



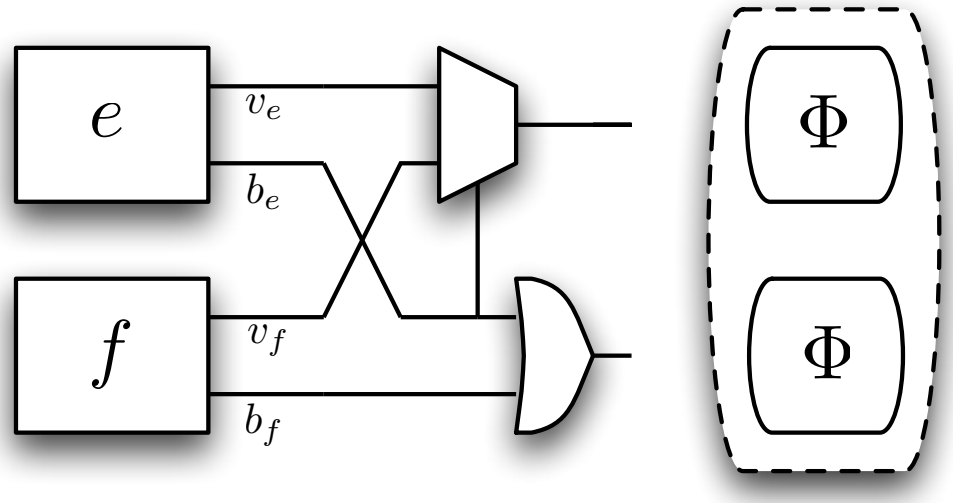
A glimpse of the first pass

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



A glimpse of the first pass

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



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

What does it look like in practice?

```
Variable var: ty → Type.
Inductive tele (A : Type): Type :=
| tend : A → tele A
| tbind : ∀ arg, expr arg →
  (var arg → tele A) → tele A.

Fixpoint compose {A B}
         (tA : tele A)
         (f : A → tele B) : tele B :=
match tA with
 | tend e ⇒ f e
 | tbind body cont ⇒
   tbind body (fun x ⇒ compose (cont x) f)
end.
```

```
fix compile {t} (a: action var ...) : tele ... :=
match a with
 | Return _ exp ⇒
    x ← ... exp;
    & (x, #b true, nil)
 | Bind _ _ e f ⇒
    compose (compile e) (fun (ve, be, Φe) ⇒
    compile (f ve) (fun (vf, bf, Φf) ⇒
    & (vf, andb be bf, [...])))
 | OrElse _ e f ⇒
    compose (compile e) (fun (ve, be, Φe) ⇒
    compose (compile f) (fun (vf, bf, Φf) ⇒
    r ← ... (vf ? ve : vf);
    & (r , (orb be bf) , [...])))
```

This is just moving binders around

Program transformations

We implement two flavors of common sub-expression elimination.

here, we focus on boolean expressions

- Incrementally build a BDD that abstracts the runtime values of variables.
- That is, we tag program variables with their abstraction.

```
Variable Var : type → Type.
Definition sval := option BDD.expr.
Notation V := (fun t ⇒ Var t * sval).
```

- Then, the main compilation function has type

```
Fixpoint cse {A} (Gamma: Env) (T : tele V A) : tele Var A := ...
```

- In short (plenty of details omitted):
  - For each (boolean) expression, build a (syntactic) boolean formula;
  - Add this formula to the global BDD pool;
  - Test whether an equivalent formula was already bound
    - if it is the case, reuse the old binder
    - if it is not the case, build a fresh binder

Program transformations

We implement two flavors of common sub-expression elimination.

here, we focus on boolean expressions

- Incrementally build a BDD that abstracts the runtime values of variables.
- That is, we tag program variables with their abstraction.

```
Variable Var : type → Type.
Definition sval := option BDD.expr.
Notation V := (fun t ⇒ Var t * sval).
```

- Then, the main compilation function has type

```
Fixpoint cse {A} (Gamma: Env) (T : tele V A) : tele Var A := ...
```

- In short (plenty of details omitted):
  - For each (boolean) expression, build a (syntactic) boolean formula;
  - Add this formula to the global BDD pool;
  - Test whether an equivalent formula was already bound
    - if it is the case, reuse the old binder
    - if it is not the case, build a fresh binder

Proving program transformations correct

- Implementing transformations in PHOAS is smooth;
- But proving them correct, we need a well-formedness property on term.
- Coming back on STLC:

$$\frac{}{(x, \tau_1, \tau_2) \in \Gamma \vdash \mathtt{Var}\ \tau_1 \equiv \mathtt{Var}\ \tau_2} \qquad \frac{\Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_1 \equiv e_1'}{\Gamma \vdash \mathtt{App}\ \mathtt{e}_1\ \mathtt{e}_2 \equiv \mathtt{App}\ \mathtt{e}_1'\ \mathtt{e}_2'}$$

$$\frac{(x, \tau_1, \tau_2) :: \Gamma \vdash e_1\ \tau_1 \equiv e_2\ \tau_2}{\Gamma \vdash \mathtt{Abs}\ \mathtt{e}_1 \equiv \mathtt{Abs}\ \mathtt{e}_2}$$

- In Coq,

```
Variables var1 var2: T → Type.
Definition Γ := list ({t: T & (var1 t * var2 t)}).
Inductive (⊢) : Γ → term var1 t → term var2 t → Prop := ...
```

Proving program transformations correct

The proof of correctness of (most) of our transformations applies only to well-formed terms.
Problem:

- For each program, it is easy to prove it WF (using tactics).
- But, we cannot prove that all programs are WF (meta-theoritical result)

Solutions:

- Either assume this parametricity property as an axiom;
- or prove that all source programs are WF (and that transformations preserve WF);
- or generate proof obligations for all intermetiate programs (and discharge them using tactics).

Wrapping-up

- Using PHOAS, it is easy to formalize EDSL in Coq.
- In this project, we use Coq's extraction to generate VHDL code from our back-end
- Proof of (intricate) circuit generators is (relatively) smooth.