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)

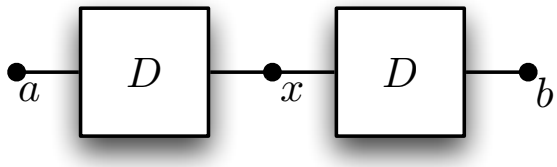
Representing circuits with predicates (or functions).

- 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)

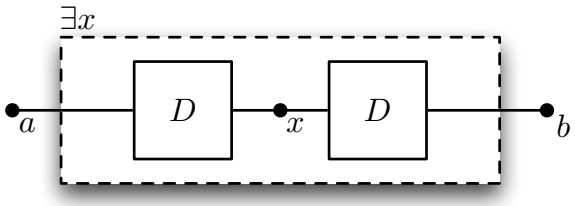
Representing circuits with predicates (or functions).

- 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:

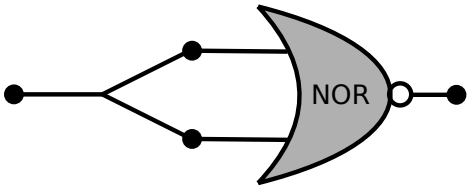
- restrict the quantification on circuits;
- reason on the structure of the circuit in the proof assistant.

Move to deep-embeddings:

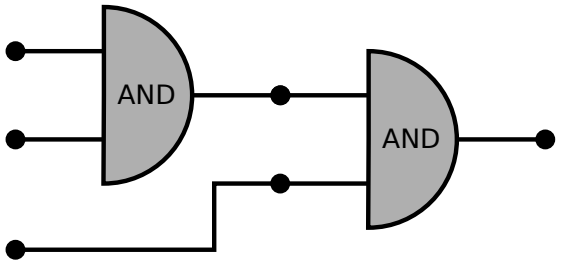
- define a data structure for circuits;
- define what's a circuit semantics (via an interpretation function);
- prove that a device implements a given specification.

No currents, unit delays

The graphical approach

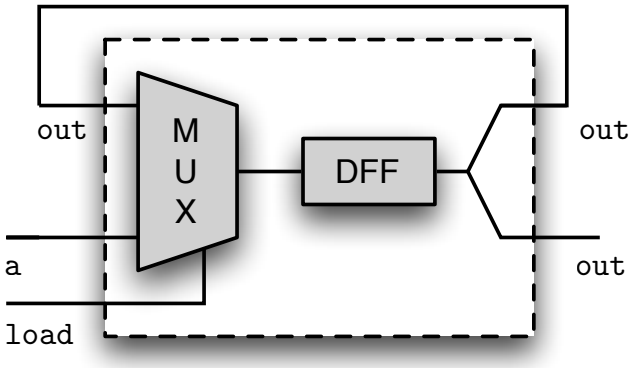



```
Fork 2 ▷ NOR
Not : circuit 1 1
```




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

The graphical approach




```
Mem : circuit 2 1
```

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's use 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's use arbitrary types as indexes for the ports:

  $\texttt{Inductive}\ \mathbb{C}\ : \texttt{Type} \to \texttt{Type} \to \texttt{Type} := \ldots$

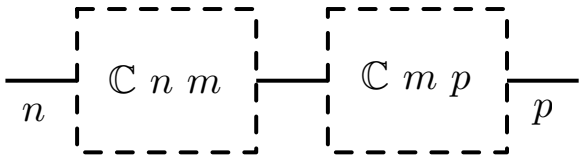
- For instance ($\mathbf{1}$ is the unit type, and $\oplus$ is disjoint-sum):

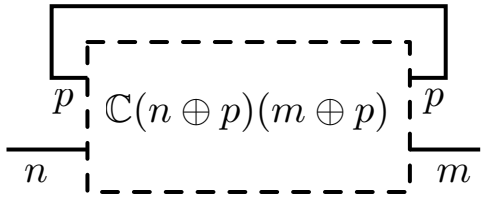
`Not`	:	$\mathbb{C}\ \mathbf{1}\ \mathbf{1}$	$\text{semantics}(\mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`And3`	:	$\mathbb{C}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1})\ \mathbf{1}$	$\text{semantics}\ (\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbf{1} \to \mathbb{B})$
`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

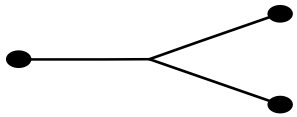
Wiring

- 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)

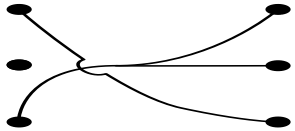
$\mathbb{C}\ (n \oplus m)\ m$

(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

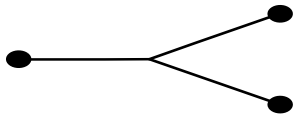
Wiring

- 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)

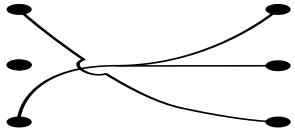
$\mathbb{C}\ (n \oplus m)\ m$

(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.
- Bad: *"This circuit implements foo* : $(\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{B})$*"*
- Better: *"This circuit implements foo* : $\mathbb{B} \otimes \mathbb{B} \to \mathbb{B} \otimes \mathbb{B}$*"*

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.
- Bad: *"This circuit implements foo* : $(\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{B})$*"*
- Better: *"This circuit implements foo* : $\mathbb{B} \otimes \mathbb{B} \to \mathbb{B} \otimes \mathbb{B}$*"*

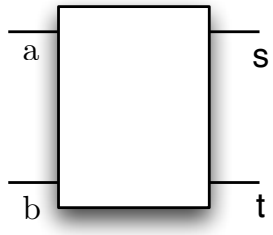
Using isomorphisms

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

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

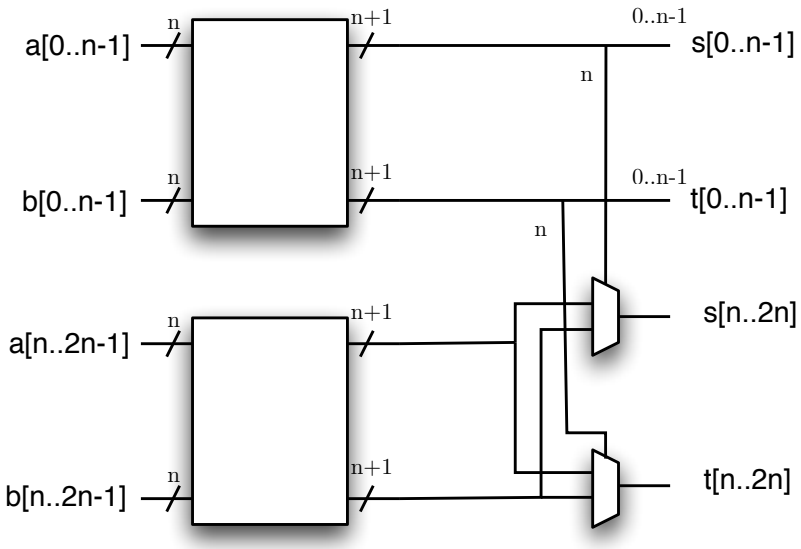
- Use it to define specifications.
- Bad: *"This circuit implements foo* : $(\mathbf{1} \oplus \mathbf{1} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{B})$*"*
- Better: *"This circuit implements foo* : $\mathbb{B} \otimes \mathbb{B} \to \mathbb{B} \otimes \mathbb{B}$*"*

A circuit generator: an adder



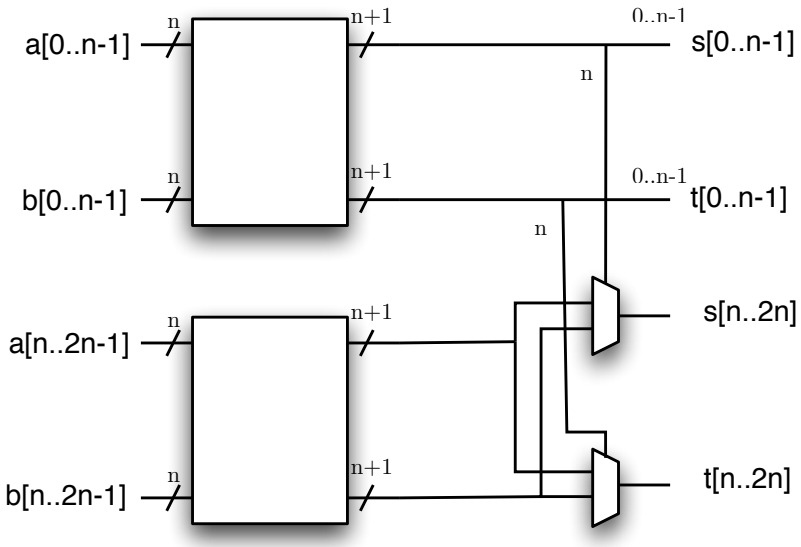
$$s = a + b \quad t = a + b + 1$$

A circuit generator: an adder



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

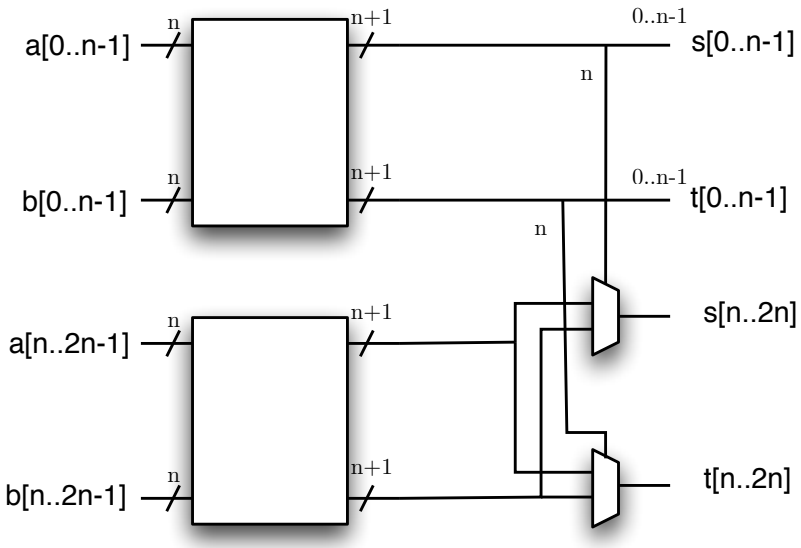
A circuit generator: an adder



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

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

A circuit generator: 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

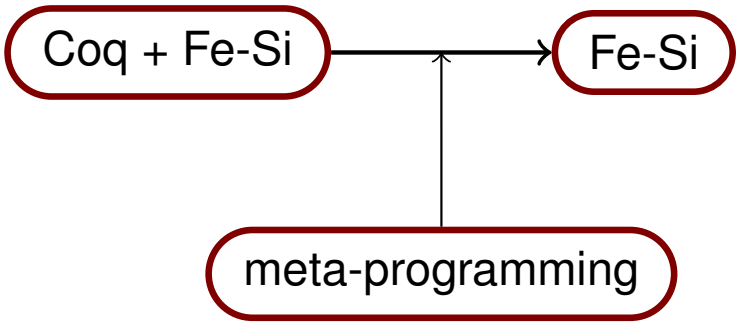
Joint work with A. Chlipala

Fe-Si (Featherweigth synthesis)

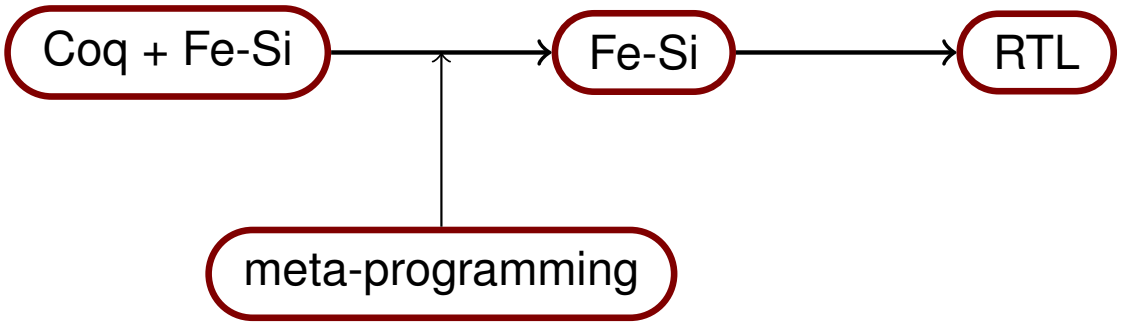
- an Embedded Domain Specific Language in Coq
- an idealized subset of the Bluespec hardware description language (MIT, industrial)
- describe circuits in “term rewriting systems” style
- semantics based on guarded atomic actions
- a “surface” language for circuit generators;
- with a compiler to a more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

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

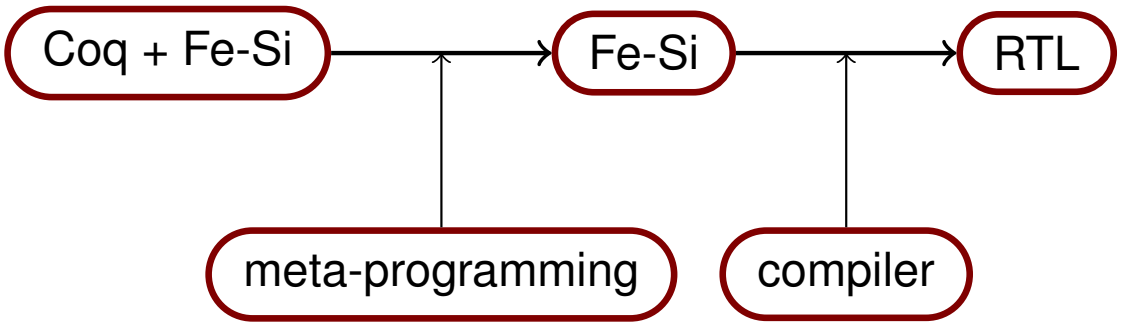
Why do we need both a compiler and meta-programming?



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

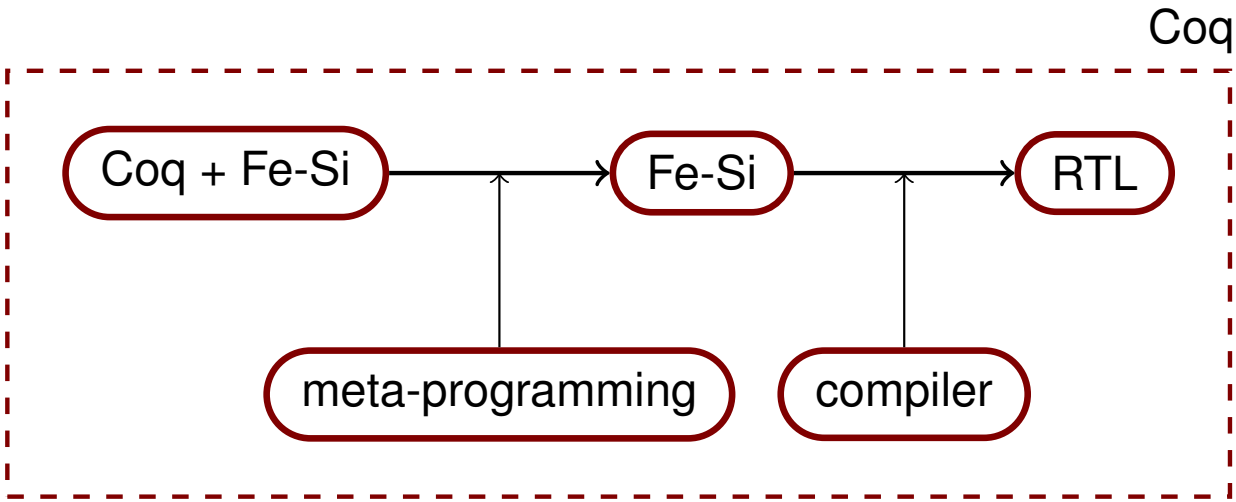


Why do we need both a compiler and meta-programming?



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

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

Fe-Si, formally

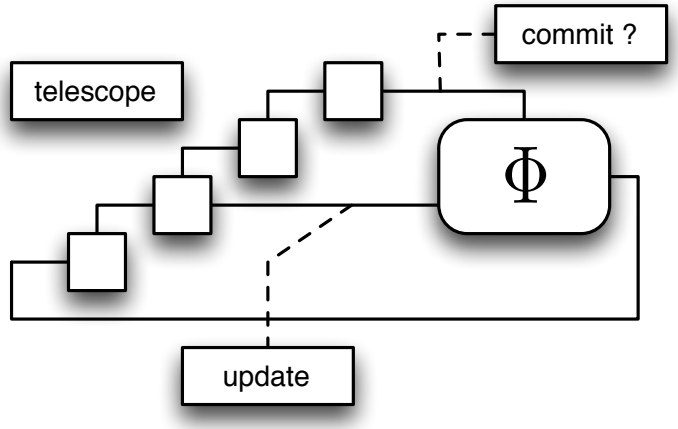
```
Variable var: ty → Type.
Inductive expr: ty → Type := ...

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

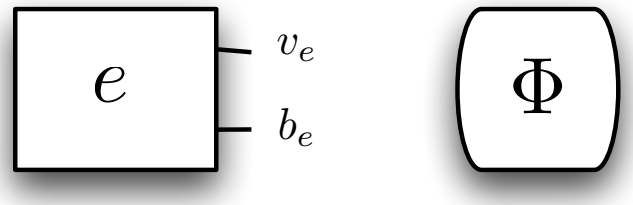
An RTL circuit is abstracted as:

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



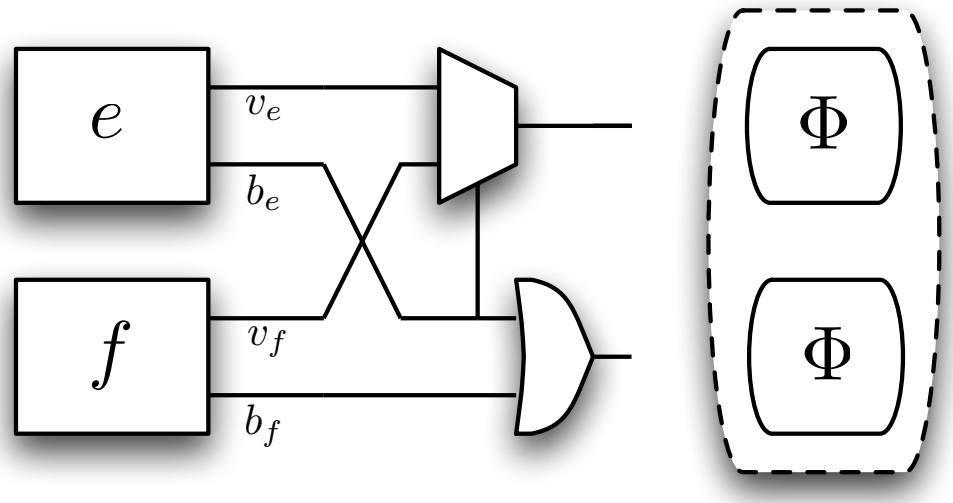
A glimpse of the first pass

- Transform control-flow into data-flow.



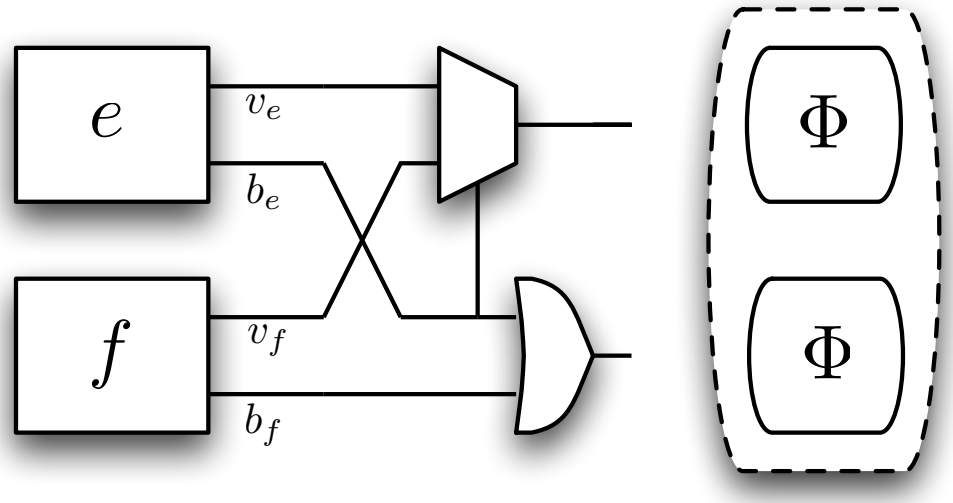
A glimpse of the first pass

- 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 ⇒
    tbind (... exp) (fun x ⇒
    tend (x, #b true, nil))
 | Bind _ _ e f ⇒
    compose (compile e) (fun (ve, be, Φe) ⇒
    compile (f ve) (fun (vf, bf, Φf) ⇒
    tend (vf, andb be bf, [...])))
 | OrElse _ e f ⇒
    compose (compile e) (fun (ve, be, Φe) ⇒
    compose (compile f) (fun (vf, bf, Φf) ⇒
    tbind (... (be ? ve : vf)) (fun r ⇒
    tend (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:

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

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

Forall term t, and variable representations ν_1, ν_2, $t\nu_1 = t\nu_2$

- 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 intermediate programs (and discharge them using tactics).

Wrapping-up

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

Coming back to circuit verification

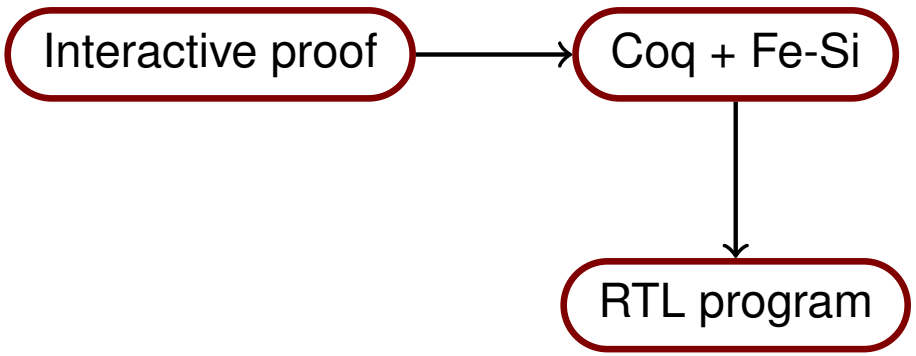
Coq + Fe-Si

- Our verification effort happens at the level of circuit generators
- It composes with the compiler proof (done once and for all)

Coming back to circuit verification

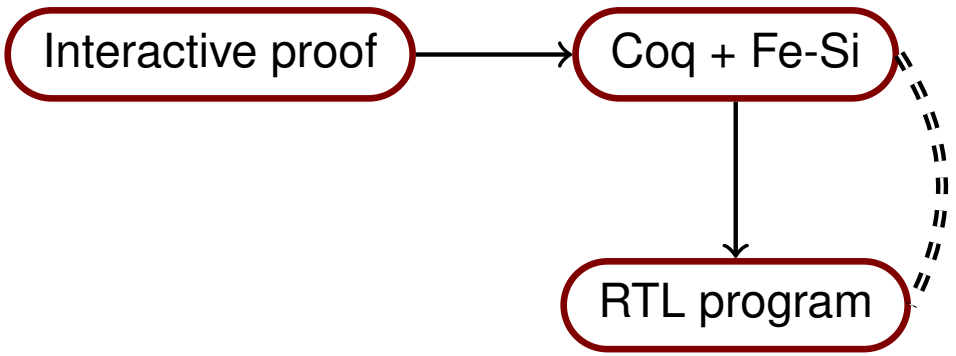
- Our verification effort happens at the level of circuit generators
- It composes with the compiler proof (done once and for all)

Coming back to circuit verification



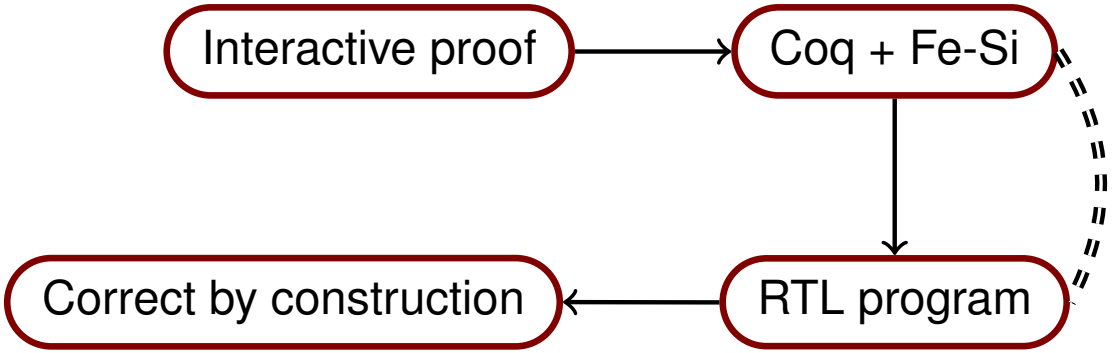
- Our verification effort happens at the level of circuit generators
- It composes with the compiler proof (done once and for all)

Coming back to circuit verification



- Our verification effort happens at the level of circuit generators
- It composes with the compiler proof (done once and for all)

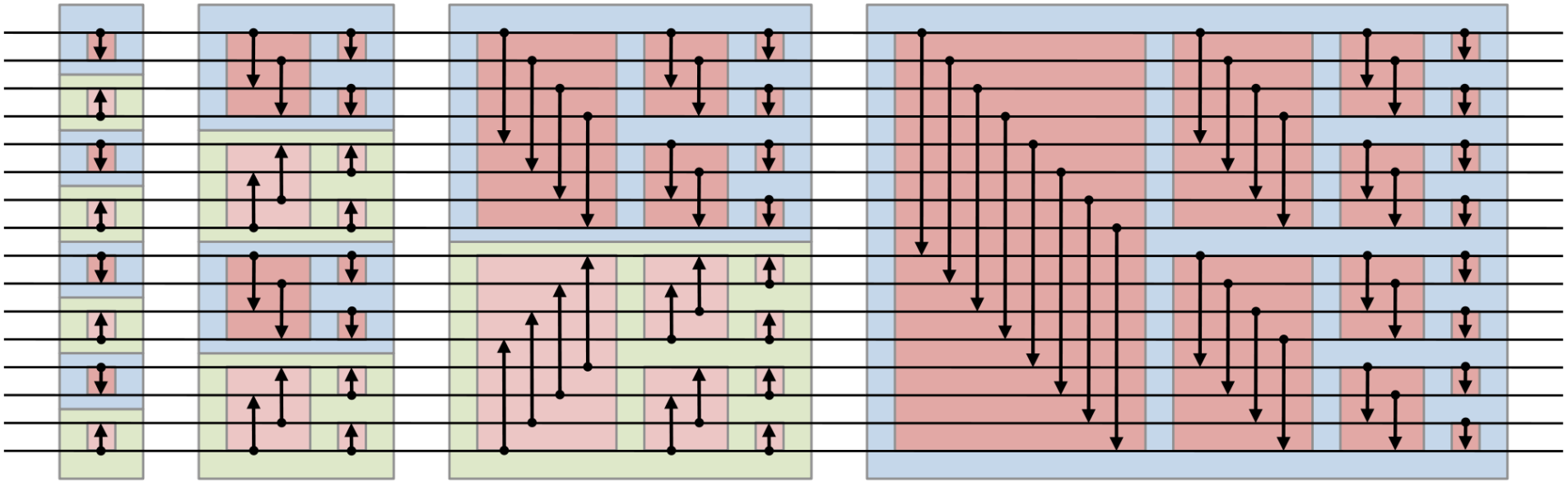
Coming back to circuit verification



- Our verification effort happens at the level of circuit generators
- It composes with the compiler proof (done once and for all)

A bitonic sorter core

- Parameters: width, type equipped with a compare-and-swap operation



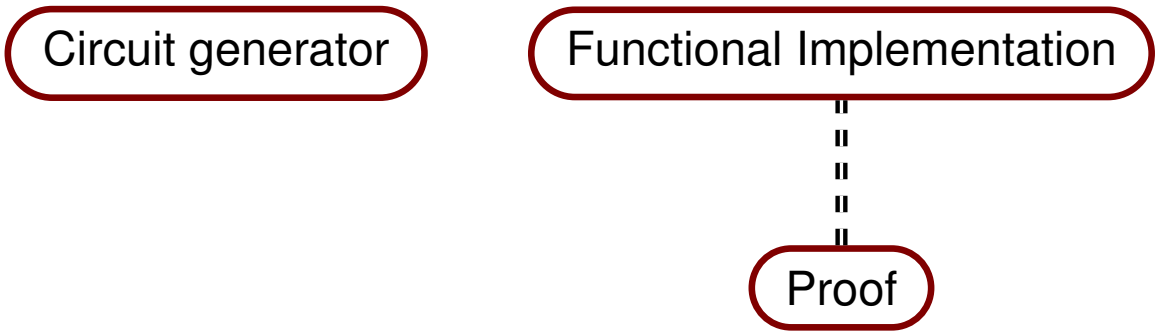
- Bitonic: $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ for some k, or a circular shift.
- Red: bitonic $\rightarrow$ l_1 bitonic, l_2 bitonic (and $l_1 \leq l_2$)
- Blue (resp. green): bitonic $\rightarrow$ sorted (resp. sorted in reverse order)

A bitonic sorter core: proof

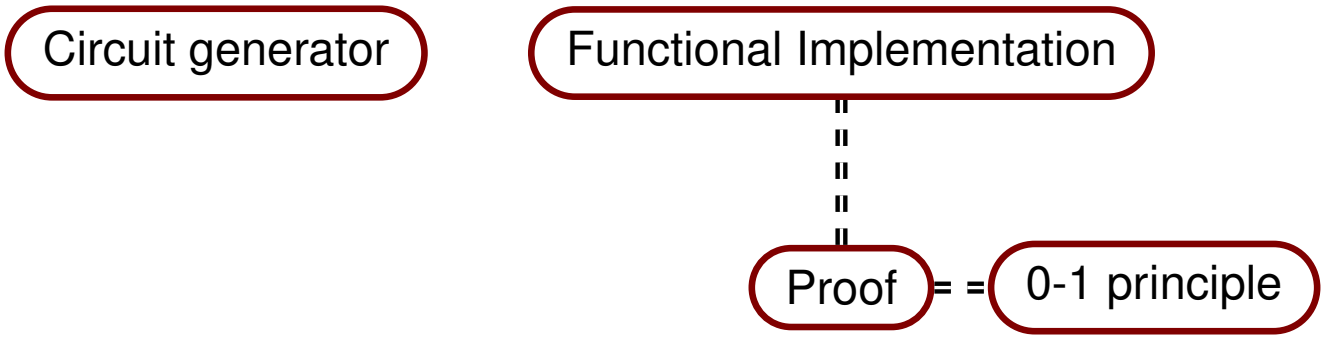
Circuit generator

Functional Implementation

A bitonic sorter core: proof



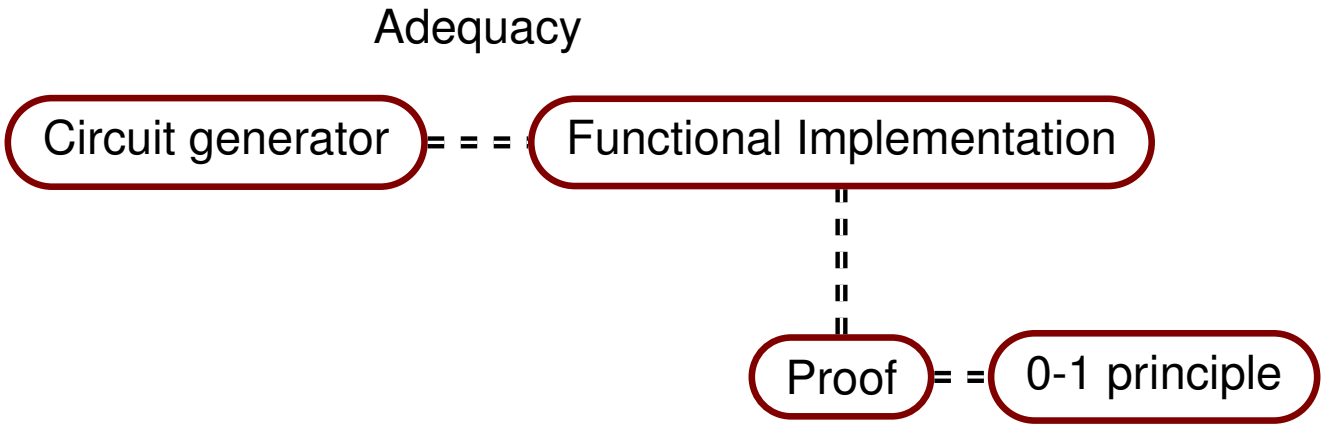
A bitonic sorter core: proof



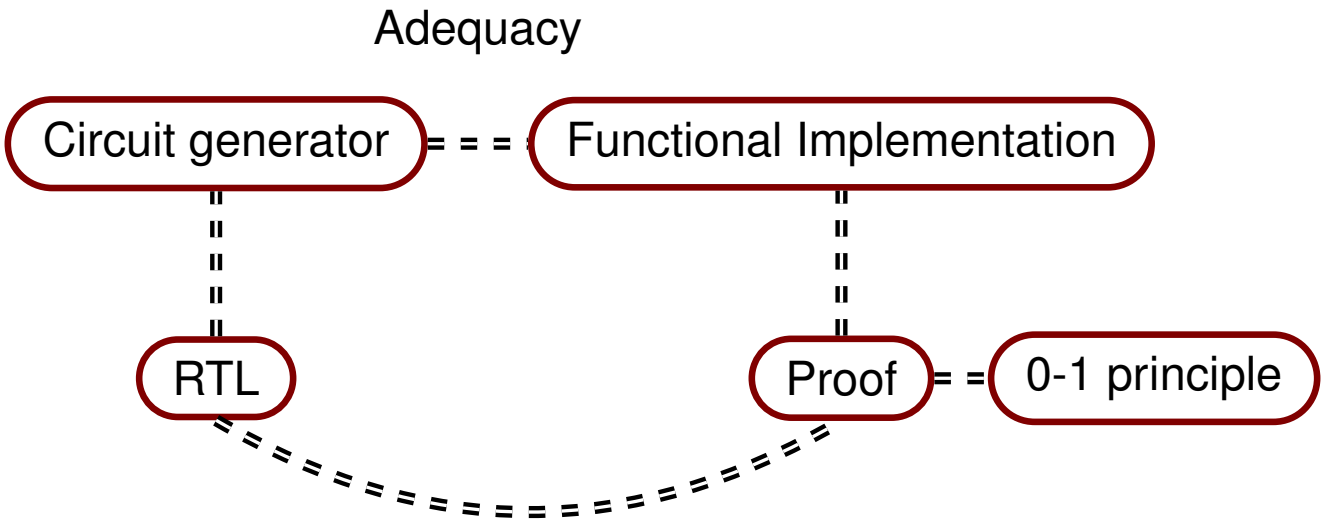
Theorem (0-1 principle)

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

A bitonic sorter core: proof



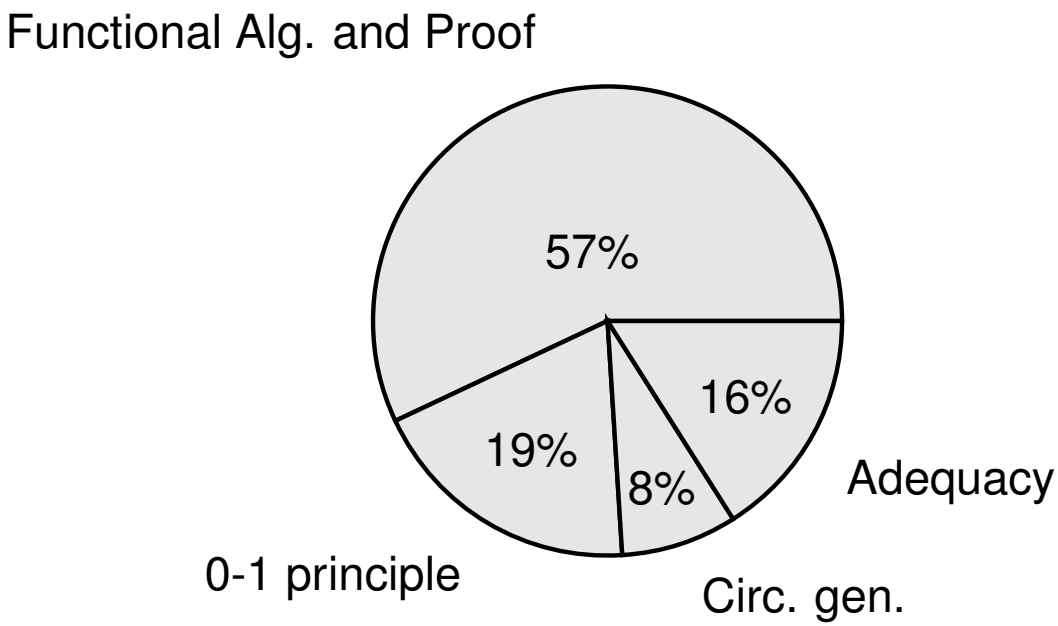
A bitonic sorter core: proof



Theorem

Given n (a bit-width), and m, we generate a circuit that sorts 2^m integers of size n.

A bitonic sorter core: verification effort



Another example: a family of stack machines

i	::=	`const` n	$\vdash pc, \sigma, s \to pc + 1, n :: \sigma, s$	
	|	`var` x	$\vdash pc, \sigma, s \to pc + 1, s(x) :: \sigma, s$	
	|	`setvar` x	$\vdash pc, v :: \sigma, s \to pc + 1, \sigma, s[x \leftarrow v]$	
	|	`add`	$\vdash pc, n_2 :: n_1 :: \sigma, s \to pc + 1, (n_1 + n_2) :: \sigma, s$	
	|	`sub`	$\vdash pc, n_2 :: n_1 :: \sigma, s \to pc + 1, (n_1 - n_2) :: \sigma, s$	
	|	`bfwd` δ	$\vdash pc, \sigma, s \to pc + 1 + \delta, \sigma, s$	
	|	`bbwd` δ	$\vdash pc, \sigma, s \to pc + 1 - \delta, \sigma, s$	
	|	`bcond` c δ	$\vdash pc, n_2 :: n_1 :: \sigma, s \to pc + 1 + \delta, \sigma, s$	if $c\ n_1\ n_2$
			$\vdash pc, n_2 :: n_1 :: \sigma, s \to pc + 1, \sigma, s$	if $\neg(c\ n_1\ n_2)$
	|	`halt`	`no reduction`	

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

For the practical minds

This was actually used to generate real Verilog code that run on real Verilog simulators

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

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

Conclusion

- A diagramatic presentation of circuits
- A idealization of Bluespec (using PHOAS)
- Used to succesfully verify parametric designs

Perspectives

- Use this diagramatic setting further
- Use singleton types to infer plugs (G. Scherer)
- Use PHOAS to embed other (non-Turing complete) languages in Coq

GPU programming anyone?

- (For fun) understand the computational content of the proof of the 0-1 principle
- Causality in circuits
- If time permits, let's organize a BDD aftertalk

Or "how we implemented four BDD libraries in Coq, and found two nifty design patterns"

Implementations of BDDs

Need to be:

- Efficient
- Verified

Problems:

- Maximal sharing
- Memoization

Requires imperative features

Implementations of BDDs

Need to be:

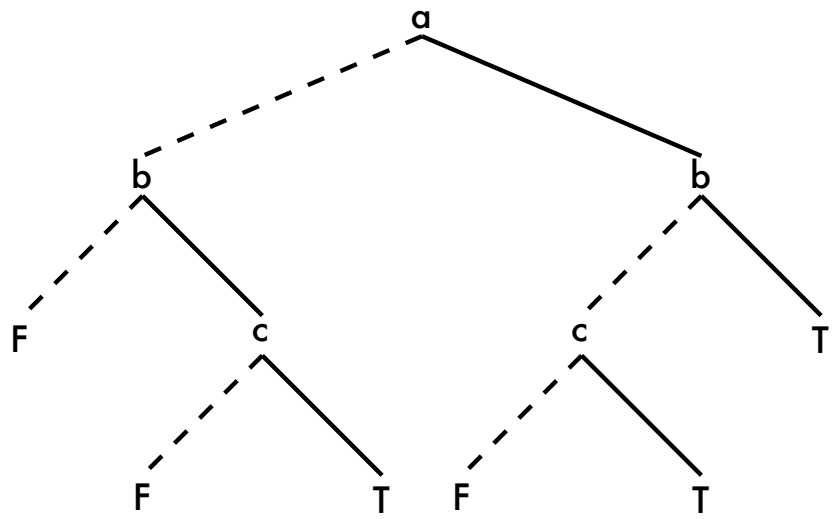
- Efficient
- Verified

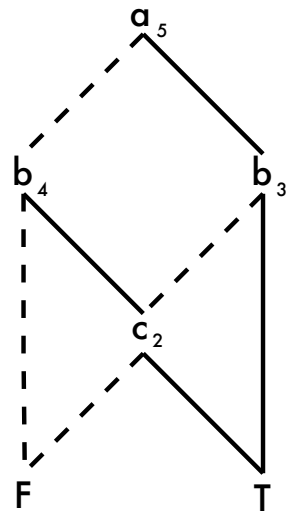
Problems:

- Maximal sharing
- Memoization

Requires imperative features

Binary decision diagrams in a nutshell



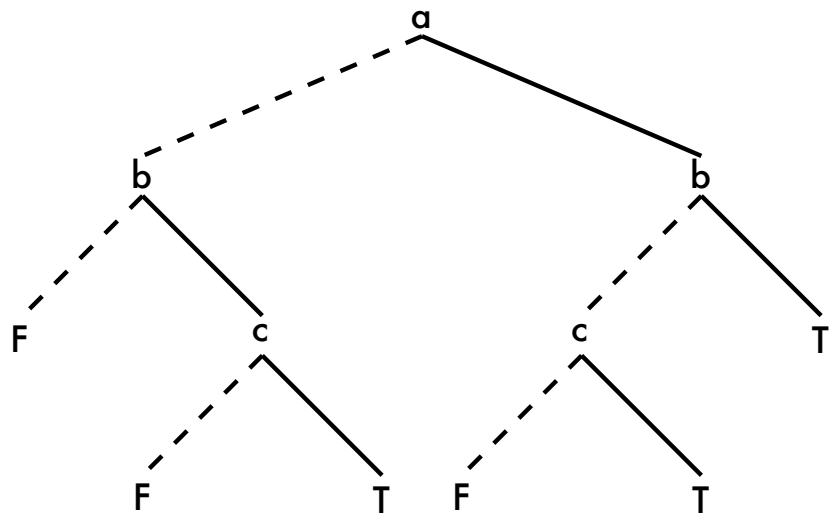


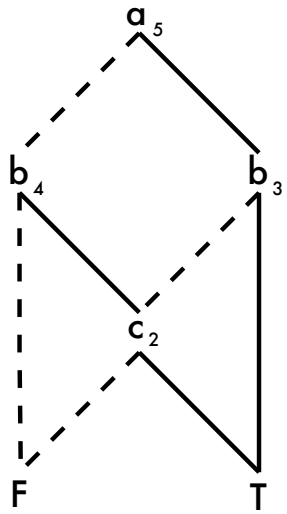
- Ensure maximum sharing by construction

In a BDD, each node is associated to a unique identifier (uid).

Build memoization table to save computations

Binary decision diagrams in a nutshell



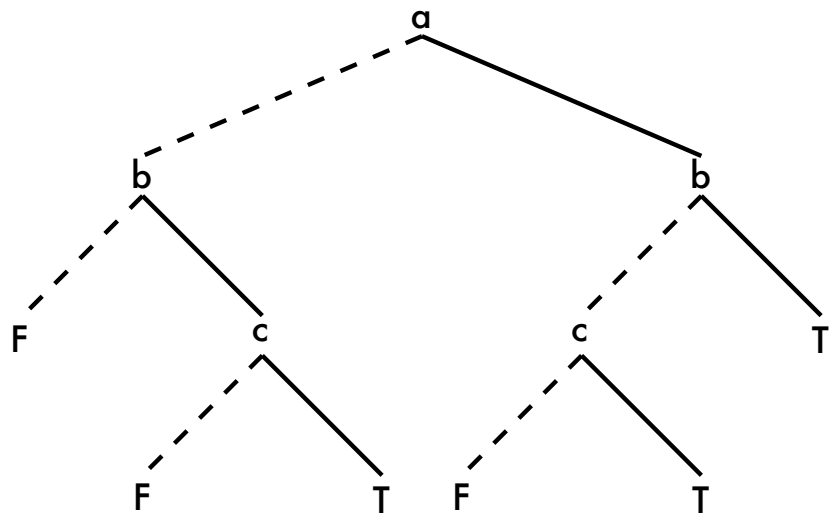


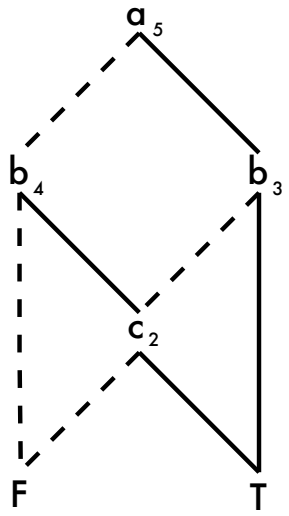
- Ensure maximum sharing by construction

In a BDD, each node is associated to a unique identifier (uid).

Build memoization table to save computations

Binary decision diagrams in a nutshell



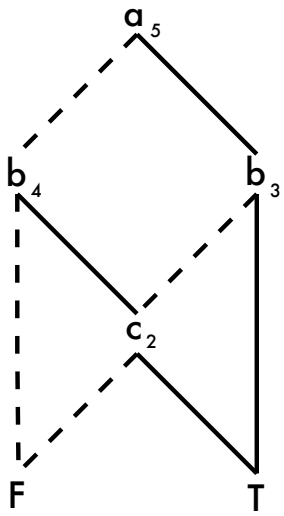


- Ensure maximum sharing by construction

In a BDD, each node is associated to a unique identifier (uid).

Build memoization table to save computations

Plan A



ENCODING OF THE GRAPH

2	↔	(F, T, c)
3	↔	(N 2, T, b)
4	↔	(F, N 2, b)
5	↔	(N 4, N 3, a)

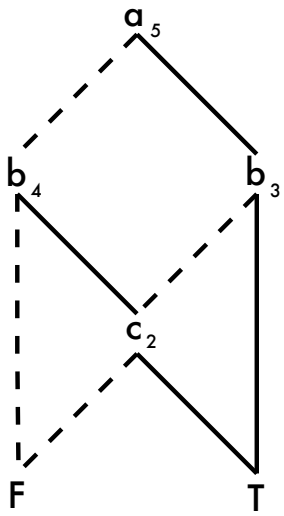
NEXT IDENTIFIER = 6

```
Notation uid := positive.
Inductive expr :=
| T: expr
| F: expr
| N: uid → expr.
Definition node := expr * expr * var.
```

Embbeding global state in a monad

- Next available uid
- Table from uid to nodes
- Table from nodes to uid
- Memoization tables

Plan A



ENCODING OF THE GRAPH

2	↔	(F, T, c)
3	↔	(N 2, T, b)
4	↔	(F, N 2, b)
5	↔	(N 4, N 3, a)

NEXT IDENTIFIER = 6

```
Notation uid := positive.
Inductive expr :=
| T: expr
| F: expr
| N: uid → expr.
Definition node := expr * expr * var.
```

Embbeding global state in a monad

- Next available uid
- Table from uid to nodes
- Table from nodes to uid
- Memoization tables

Discussing Plan A

Invariants:

- Both table form a bijection
- All used identifiers bellow next identifier
- Memoization tables contain only valid information
- State is monotone

Have to prove them for every operation

Discussing Plan A

Drawbacks:

- Monadic interface
- Not modular: need to modify the monad to add features

e.g. memoizing a new operator

- Non-structural recursion
  - Well-founded recursion not easy here

`Program Fixpoint` and `Function` not adapted

  - Need some fuel ?
- No easy garbage collection

Plan B

```
Inductive expr :=
| F
| T
| N : hc_expr → hc_expr → var → expr
with hc_expr :=
HC : expr → positive → hc_expr.
```

Embbeding global state in a monad

- Next available uid
- Table from `expr` to `hc_expr`
- Memoization tables

Discussing Plan B

Invariants:

- All used identifiers bellow next identifier
- Memoization tables contain only valid information
- State is monotone

Have to prove them for every operation

Discussing Plan B

Drawbacks:

- Monadic interface
- Not modular: need to modify the monad to add features

e.g. memoizing a new operator

- Slightly easier garbage collection

Plan C

No sharing in Coq code

```
Inductive bdd: Type :=
| T: bdd
| F: bdd
| N: var → bdd → bdd → bdd.
```

Extract N to a hash-consing constuctors

In pseudo-OCaml:

```
let smartN var nT nF =
 try HcTable.find (var, nT, nF)
 with Not_found →
  let node = N var nT nF in
  HcTable.add (var, nT, nF) node;
  node
```

- Weak hash tables
- GC for free

Memoization without monad

Memoizing fixpoint combinator

- Extracted to specialized OCaml code.
- Well-founded to tackle complex recursion schemes.

```
Definition memoFix1 := Fix (well_founded_ltof bdd bdd_size).

Extract Inlined Constant memoFix1 ⇒ "Helpers.memoFix1".

Program Definition bdd_not : bdd → bdd :=
 memoFix1 (fun b rec ⇒
      match b with
       | T ⇒ F  | F ⇒ T
       | N v bt bf ⇒ N v (rec bt) (rec bf)
      end).
```

Plan D: Exposing unique identifiers

Could be needed as indices to tables

- Add an abstract type for unique identifiers
- Axiomatizing the bijection between unique identifiers and bdd

Conclusion

Clear winners: plan B and C

- Smart constructors are a breeze (proofs, efficiency after extraction, gc)
- Plan B and C used to implement strong normalization in the lambda-calculus
- Submitted to JAR
- Question: automatic smart extraction in Coq?