Compiling Coq in Coq

Gregory Malecha
gmalecha@cs.harvard.edu

Harvard SEAS

January 17, 2013

Programming Languages

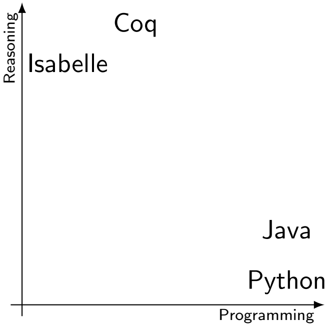
- Coq is predominantly a reasoning language!
    - 4-color theorem
    - Feit Thompson
    - PL Meta-theory
    - Topology
    - **CompCert**

Programming Languages

- Coq is predominantly a reasoning language!
  - 4-color theorem
  - Feit Thompson
  - PL Meta-theory
  - Topology
  - **CompCert**
- What about:
  - Hedge-fund trading algorithms?
  - Web applications?
  - ...

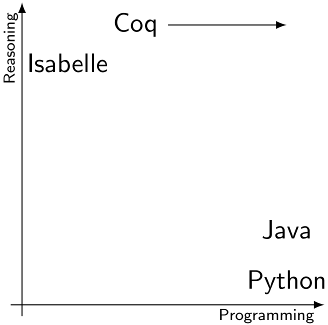
Programming Languages

- Coq is predominantly a reasoning language!
  - 4-color theorem
  - Feit Thompson
  - PL Meta-theory
  - Topology
  - **CompCert**
- What about:
  - Hedge-fund trading algorithms?
  - Web applications?
  - ...



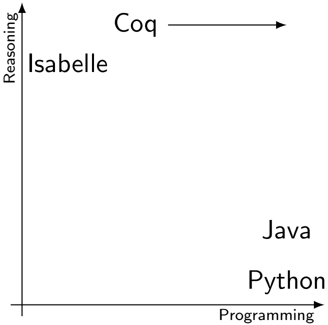
Programming Languages

- Coq is predominantly a reasoning language!
  - 4-color theorem
  - Feit Thompson
  - PL Meta-theory
  - Topology
  - **CompCert**
- What about:
  - Hedge-fund trading algorithms?
  - Web applications?
  - ...



Programming Languages

- Coq is predominantly a reasoning language!
  - 4-color theorem
  - Feit Thompson
  - PL Meta-theory
  - Topology
  - **CompCert**
- What about:
  - Hedge-fund trading algorithms?
  - Web applications?
  - ...



Coq as a **programming** language!

Programming Features?

- Some features of a **programming** language
  - Execution/compilation
  - I/O
  - Libraries
  - Debugging
- How to reason about *some* of them

Programming Features?

- Some features of a **programming** language
  - Execution/**compilation**
  - I/O
  - Libraries
  - Debugging

Case Study

- How to reason about *some* of them

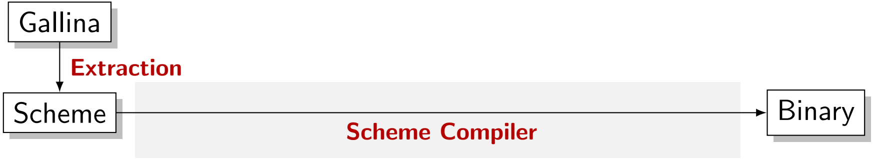
Outline

1. Building the Compiler
   - cs252: The Class
   - The Artifact
2. Coq Programming Library
   - Dependent Types, Prop & Computation
   - Programming with Monads
   - Notation
   - Type Class Resolution
3. What's Next?
   - Compiler
   - ExtLib

Outline

1. Building the Compiler
   - cs252: The Class
   - The Artifact
2. Coq Programming Library
   - Dependent Types, Prop & Computation
   - Programming with Monads
   - Notation
   - Type Class Resolution
3. What's Next?
   - Compiler
   - ExtLib

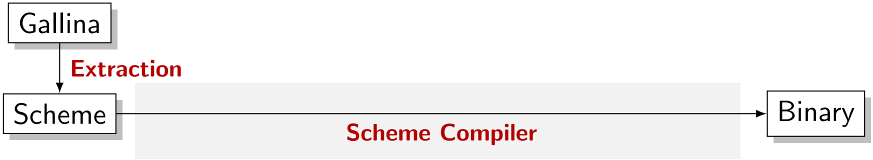
Aside: Why not just extraction?



Extraction

- Leverages existing technology (OCaml, Scheme, Haskell)
- Fast code
- Relatively easy

Aside: Why not just extraction?



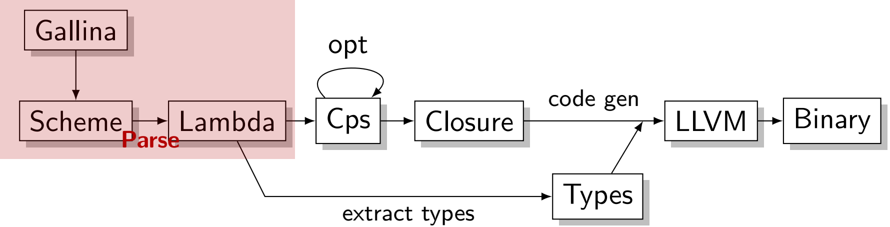
Extraction

- Leverages existing technology (OCaml, Scheme, Haskell)
- Fast code
- Relatively easy

Compilation

- **Learning** about compilers
- **Programming** in Coq
- Optimization potential

cs252r: The Course

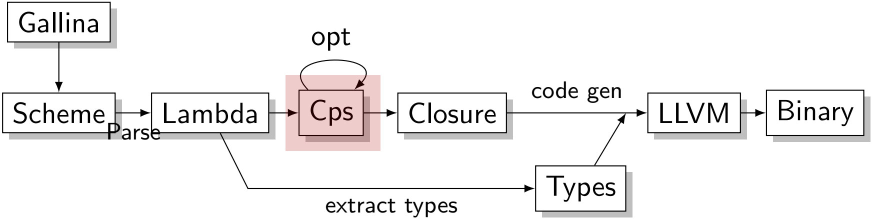


0. **Setup**

Lambda

- Based on extraction IL
- Single level `match`
- First-order binders
- Recursive `let`

cs252r: The Course

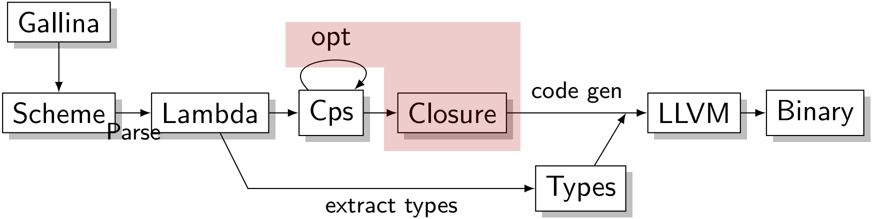


0. **Setup**
1. Warm-up **Cps Interpreter**

Cps

- Interpreter for Cps representation
- Constructors → tuples
- `match` → `switch`

cs252r: The Course

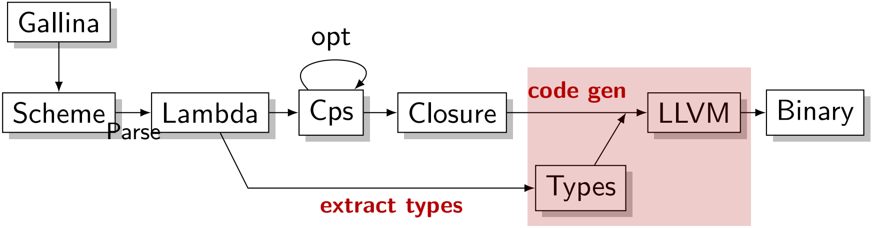


0. **Setup**
1. Warm-up **Cps Interpreter**
2. **Closure Conversion & Opt**

Closure Conversion

- Common subexpression elimination
- Closure conversion

cs252r: The Course

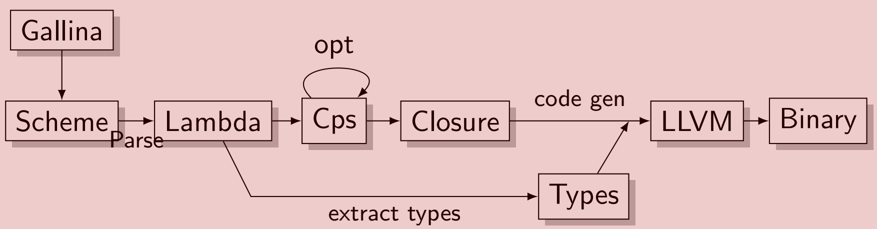


0. **Setup**
1. Warm-up **Cps Interpreter**
2. **Closure Conversion & Opt**
3. **Code Generation**

Code Generation & Runtime

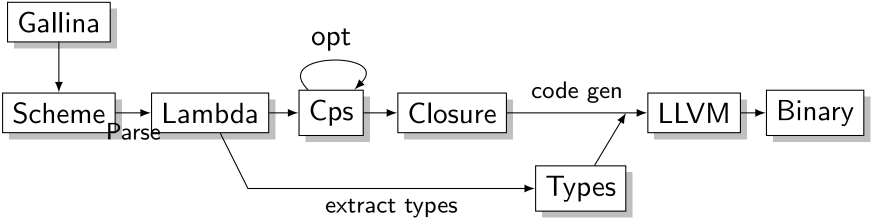
- Lower to LLVM
- Allocate construtor tags
  - Extract types from Lambda
- Garbage collection

cs252r: The Course



0. **Setup**
1. Warm-up **Cps Interpreter**
2. **Closure Conversion & Opt**
3. **Code Generation**
4. **Final Project** ...

cs252r: Final Projects

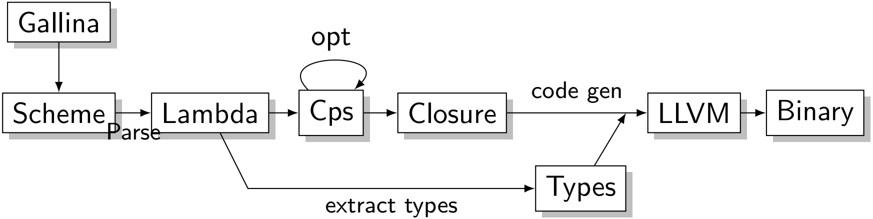


4 group projects

1. Core Compilation, abstract interpretation, runtime
2. Optimization: Inlining, unboxing, uncurrying
3. Optimization: Value Irrelevance[1]
4. Reading: Abstract Interpretation

[1] Different compilation framework

cs252r: Final Projects

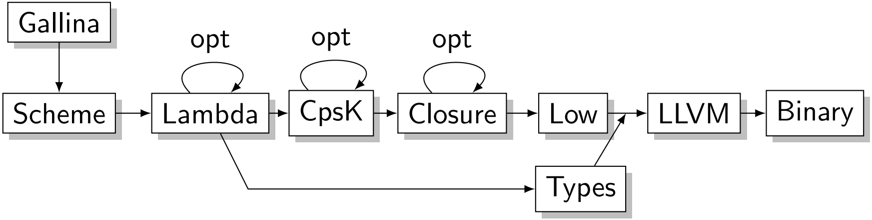


4 group projects

1. **Core Compilation, abstract interpretation, runtime**
2. Optimization: Inlining, unboxing, uncurrying
3. Optimization: Value Irrelevance[1]
4. Reading: Abstract Interpretation

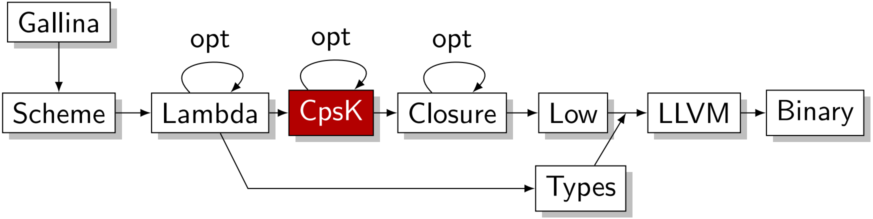
[1] Different compilation framework

The Artifact



- Refactored Cps IR, added I/O
- Fixed lots of bugs (debugging)
- Wrote compilation wrapper (command line tool)

CpsK



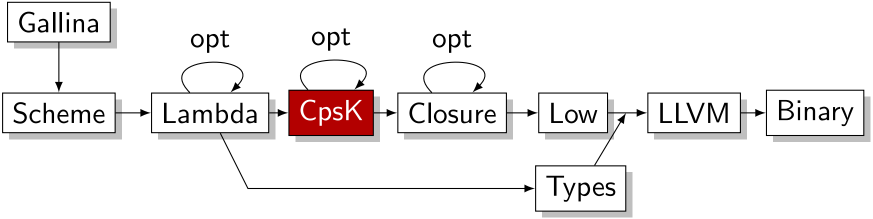
CpsK

- Second class continutations
  - Stack-allocatable
- Monadic operations (I/O)
  - World-passing
  - Supports functional CSE
- Contification optimization

Second Class Continuations

```
Inductive cps : Type :=
| Let  : var → list var → cps → cps
| App  : op → list op → cps
| LetK : cont → list var → cps → cps
| AppK : cont → list op
```

CpsK



CpsK

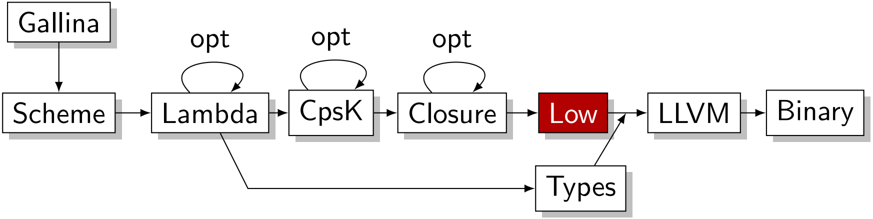
- Second class continutations
  - Stack-allocatable
- Monadic operations (I/O)
  - World-passing
  - Supports functional CSE
- Contification optimization

Monadic I/O

```
(** IO t = (t → world → ⊥) → world → ⊥
 **
 ** PrintChar :: ascii → IO unit
 ** fun k a ⇒
 **   let res = fun k' w ⇒
 **     let (x,w') = bind PrintChar (w::a::nil)
 **     in k' x w'
 **   in k res
 **)
```

Explicit world

Low



Low

- Imperative (world erased)
- Similar to basic blocks
- Intermediate code gen
- Destructive update optimization

Destructive Update

```
let x := (1,2) in
...
let y := (4, 3) in
(* x not used *)
```

Reuse "dead" memory

Optimizes the state monad!

Low vs SSA

Low

```
int foo(int a) {
entry(a):
  int b = 0;
  goto top(a, b)
top(a_1,b_1):
  if a_1 = 0 { goto bot(b_1) }
       else { goto loop(a_1,b_1) }
loop(a_2,b_2):
  int a_3 = a_2 − 1;
  int b_3 = b_2 + 1;
  goto top(a_3,b_3)
bot(b_4):
  return (b_4)
}
```

Block arguments

Block "calls"

SSA (LLVM)

```
int foo(int a) {
entry:
  int b = 0;
  goto top(a, b)
top:
  a_1 = φ(a,a_2)
  b_1 = φ(b,b_2)
  if a_1 = 0 { goto bot }
       else { goto loop }
loop:
  int a_2 = a_1 − 1;
  int b_2 = b_1 + 1;
  goto top
bot:
  return b_1
}
```

Low vs SSA

Low

```
int foo(int a) {
entry(a):
  int b = 0;
  goto top(a, b)
top(a_1,b_1):
  if a_1 = 0 { goto bot(b_1) }
        else { goto loop(a_1,b_1) }
loop(a_2,b_2):
  int a_3 = a_2 − 1;
  int b_3 = b_2 + 1;
  goto top(a_3,b_3)
bot(b_4):
  return (b_4)
}
```

Lexical variable scope!

Local liveness info

SSA (LLVM)

```
int foo(int a) {
entry:
  int b = 0;
  goto top(a, b)
top:
  a_1 = φ(a,a_2)
  b_1 = φ(b,b_2)
  if a_1 = 0 { goto bot }
        else { goto loop }
loop:
  int a_2 = a_1 − 1;
  int b_2 = b_1 + 1;
  goto top
bot:
  return b_1
}
```

Low vs SSA

Low

```
int foo(int a) {
entry(a):
  int b = 0;
  goto top(a, b)
top(a_1,b_1):
  if a_1 = 0 { goto bot(b_1) }
        else { goto loop(a_1,b_1) }
loop(a_2,b_2):
  int a_3 = a_2 - 1;
  int b_3 = b_2 + 1;
  goto top(a_3,b_3)
bot(b_4):
  return (b_4)
}
```

Not hard to convert to SSA

SSA (LLVM)

```
int foo(int a) {
entry:
  int b = 0;
  goto top(a, b)
top:
  a_1 = φ(a,a_2)
  b_1 = φ(b,b_2)
  if a_1 = 0 { goto bot }
        else { goto loop }
loop:
  int a_2 = a_1 - 1;
  int b_2 = b_1 + 1;
  goto top
bot:
  return b_1
}
```

Runtime[2]

- Very simple copying garbage collector
  - Uses LLVM's shadow-stack to track GC roots (heavy-weight)
  - Over-aggressive LLVM optimization requires spill slots to be marked volatile
- Only support code-gen for a single continuation
  - Continuation call is compiled to a `return`
  - No multi-level returns

[2] Implemented by Scott Moore

The Compiler: Stats

- >10,000 lines of Coq code
  - Almost entirely executable code
- 1 prof + ∼4 grad students + 2 undergrads
  - Scott Moore, Dan Huang, Gregory Malecha, Greg Morrisett
  - Lucas Waye, Carl Jackson, Gabby Ehrlich
- ∼13 week course (∼4 weeks on project)
- Open source: `https://github.com/coq-ext-lib/coq-compile`

Results

- Can compile a few programs
  - Mostly difficulty with scaling at the moment

Program	Code Size
HelloWorld	91
IONat	1323
IOFact	1359
Compiler[3]	47579

- All programs use a sophisticated `Show` type class to text
  - `nat_to_string` is about 1300 lines
  - Exercises most, if not all, of the compiler

Enough to uncover an extraction bug

[3] Stack overflow

Outline

Experience

- Most students just learning Coq
  - Easier to program using
  - Avoid sophisticated features

[4] `https://github.com/coq-ext-lib/coq-ext-lib`

Experience

- Most students just learning Coq
  - Easier to program using
  - Avoid sophisticated features
- Built a new library, **ExtLib**[4], focused on **programming**
  1. After-the-fact verification
     - With generic automation
  2. First-class abstractions
     - Heavily type-class oriented
     - Avoid modules
  3. **Not full featured!**
     - Not a replacement for standard library
     - Explore different interfaces

[4] `https://github.com/coq-ext-lib/coq-ext-lib`

1. Building the Compiler
   - cs252: The Class
   - The Artifact
2. Coq Programming Library
   - Dependent Types, Prop & Computation
   - Programming with Monads
   - Notation
   - Type Class Resolution
3. What's Next?
   - Compiler
   - ExtLib

Relations and their Decision Procedures

- I like to reason about `Prop`
- Programs decide propositions (specifications)
  - Reasoning/verification is making the connection

Simple Types

```
Definition beq_nat :
  nat → nat → bool.
Theorem beq_nat_eq : ∀ a b,
  beq_nat a b = true ↔ a = b.
```

✓ Easy to write

✗ Tedious to reason about

Dependent Types

```
Definition nat_dec :
  ∀ a b : nat, {a = b} + {a ≠ b}.
```

✓ Easy to reason about

✗ Hard to write

✗ Slow computation

Type-classes for easy, after-the-fact Verification

- **Idea #1**: Lookup functions with relations!

Looking up Functions

```
Class RelDec {T} (R : relation T) :=
  rel_dec : T → T → bool.
Notation "a ?[ R  ]  b" := (@rel_dec _ R _ a b).
... if  a ?[ eq ] b then ...  else ...
```

- **Idea #2**: Lookup proofs using functions!

Looking up Functions

```
Class RelDec_Ok {T R} (RD : RelDec R) :=
  rel_dec_Ok : ∀ a b, rel_dec a b = true ↔ R a b.
Ltac case_split :=
  match goal with
    | ⊢ context [ @rel_dec _ ?R ?RD ?A ?B ] ⇒
        let pf := constr:(_ : @RelDec_Ok _ R RD) in (** apply rel_dec_Ok **)
  end.
```

Convenience with Monads

Convention for using monads

- **Easy to write**
- Huge "bang for the buck"
  - Use without really understanding
  - Used almost everywhere in the compiler

```
Section monadic.
  Variable m : Type → Type.
  Context {M : Monad m}.
  Context {MS : MonadState nat m}.

  Definition fresh : m nat :=
    x ← get ;;
    put (S x) ;;
    ret x.
End monadic.
```

Convenience with Monads

Convention for using monads

- **Easy to write**
- Huge "bang for the buck"
  - Use without really understanding
  - Used almost everywhere in the compiler

- **...Harder to run**
- Sometimes difficult to instantiate multiple monads
  - Lots of type-class resolution
  - Very complex error messages
  - Difficult to explain

```
Section monadic.
  Variable m : Type → Type.
  Context {M : Monad m}.
  Context {MS : MonadState nat m}.

  Definition fresh : m nat :=
    x ← get ;;
    put (S x) ;;
    ret x.
End monadic.

Definition fresh' : state (nat*nat)
  := (* m inferred from state *)
  a ← fresh ;;
  b ← fresh ;;
  ret (a,b).
```

Abstract Monad Reduction

- Use equational laws for tactic-based, abstract reduction!

```
Section monadic.
  Variable m : Type → Type.
  Variable M : Monad m.

  Class MonadLaws : Type :=
  { bind_of_return : ∀ A B a f,
    bind (ret a) f = f a
  ; return_of_bind : ∀ A aM f,
    (∀ x, f x = ret x) →
    bind aM f = aM
  ; bind_associativity : ...
  }.
End monadic.
Hint Rewrite bind_of_return
     bind_associativity : monad_rw.

Ltac monad_reduce := ...
```

```
Section reasoning.
  Variable m : Type → Type.
  Context {M : Monad m}.
  Context {MS : MonadState nat m}.
  Context {MOk : MonadLaws M}.
  Context {MSOk : MonadStateLaws MS}.

  Goal x ← ret 1 ;; y ← ret 2 ;; ret
     (x + y) = ret 3.
    monad_reduce. reflexivity.
  Qed.

  Goal put 3 ;; get = put 3 ;; ret 3.
    monad_reduce. reflexivity.
  Qed.
```

Notation

- Conservative about notation.

Pros

- Easier to read
- "Mathematical" presentation
- Control precedence/avoid parentheses

Cons

- Interpretation scopes are annoying
- Notation clashes prevent module use!
- Can be expensive

Mult.v:

```
  Definition mult := ....
  Infix "*" := mult (at level 40).
```

Separation.v:

```
  Definition star := ....
  Infix "*" := star (at level 52).
```

```
Require Import Mult.
Require Import Separation. (* XXX *)
```

Notation

- Conservative about notation.

Pros

- Easier to read
- "Mathematical" presentation
- Control precedence/avoid parentheses

Cons

- Interpretation scopes are annoying
- Notation clashes prevent module use!
- Can be expensive

```
Mult.v:
  Definition mult := ....
  Infix "*" := mult (at level 40).
Separation.v:
  Definition star := ....
  Infix "*" := star (at level 52).
```

```
Require Import Mult.
Require Import Separation. (* XXX *)
```

ExtLib convention

All notation in sub modules.

Type Classes for Notation

- **Idea**: Associate notation with type class functions[5]

```
Class Mult (T : Type) : Type :=
  mul : T → T → T.
Notation "x * y" := (@mul _ _ x y).

Instance Mult_Type : Mult Type:=prod.
Instance Mult_nat : Mult nat:=mul.
Instance Mult_Z : Mult Z:=Z.mul.

Definition test a b c d : nat * Z :=
  (a * b, c * d).
```

Cons

✓ Works in a lot of places

[5] Proposed by David Darais.

Type Classes for Notation

- **Idea**: Associate notation with type class functions[5]

```
Class Mult (T : Type) : Type :=
  mul : T → T → T.
Notation "x * y" := (@mul _ _ x y).

Instance Mult_Type : Mult Type:=prod.
Instance Mult_nat : Mult nat:=mul.
Instance Mult_Z : Mult Z:=Z.mul.

Definition test a b c d : nat * Z :=
  (a * b, c * d).

(* doesn't work with constants *)
Definition oops : nat * Z :=
  (1 * 2, 4 * 5).
```

Cons

- ✓ Works in a lot of places
- ✗ Does not work everywhere
- ✗ More Δ reductions
  - Can be expensive

[5] Proposed by David Darais.

Type Classes for Notation

- **Idea**: Associate notation with type class functions[5]

```
Class Mult (T : Type) : Type :=
  mul : T → T → T.
Notation "x * y" := (@mul _ _ x y).

Instance Mult_Type : Mult Type:=prod.
Instance Mult_nat : Mult nat:=mul.
Instance Mult_Z : Mult Z:=Z.mul.

Definition test a b c d : nat * Z :=
  (a * b, c * d).

(* doesn't work with constants *)
Definition oops : nat * Z :=
  (1 * 2, 4 * 5).
```

Cons

- ✓ Works in a lot of places
- ✗ Does not work everywhere
- ✗ More Δ reductions
  - Can be expensive
- ? Easy to reason about
- ? Scalable

[5] Proposed by David Darais.

1. Building the Compiler
   - cs252: The Class
   - The Artifact

2. Coq Programming Library
   - Dependent Types, Prop & Computation
   - Programming with Monads
   - Notation
   - Type Class Resolution

3. What's Next?
   - Compiler
   - ExtLib

Type Class Resolution

- Type class resolution is very convenient

Type Class Resolution

- Type class resolution is very convenient
  ...but it can be un-intuitively expensive.
- Example: Report errors from parser
  - Changing `ret None` to `raise "error"` added 3 minutes to compilation!

All instances are immediates!

Type Class Resolution

- Type class resolution is very convenient
  ...but it can be un-intuitively expensive.
- Example: Report errors from parser
  - Changing `ret None` to `raise "error"` added 3 minutes to compilation!

All instances are immediates!

- Error messages can be difficult to understand
  - Mainly when instances depend on each other
  - `Typeclasses eauto` := debeug is very helpful

Outline

1. Building the Compiler
   - cs252: The Class
   - The Artifact
2. Coq Programming Library
   - Dependent Types, Prop & Computation
   - Programming with Monads
   - Notation
   - Type Class Resolution
3. What's Next?
   - Compiler
   - ExtLib

What's Next?

Still lots to do.

- Compiler
  - **Scaling**! Be able to compile the compiler
  - Modular compilation
  - Leveraging Coq

Extended Optimization

- Recent work on extending optimizers[6]
- We already have the semantics & a language for reasoning
- Encode rewrites as records

```
Class OptEquation T := OptRewrite
{ env   : list Type
; left  : (∀ n, nth n env unit) → T
; right : (∀ n, nth n env unit) → T
; proof : ∀ g, left g = right g
}.

(** Example rewrite **)
Instance Eqn_hd T : OptEquation T :=
{ env     := T :: list T :: T :: nil
; left  g := hd (g 2) (cons (g 0) (g 1))
; right g := g 0
}. reflexivity.
Defined.
```

[6] Tate'10

Extended Optimization

- Recent work on extending optimizers[6]
- We already have the semantics & a language for reasoning

- Encode rewrites as records
- When can we rewrite?
  - Lambda? Cps? Clo?
  - ✓ More opportunities
  - ✗ More unpredictable

```
Class OptEquation T := OptRewrite
{ env   : list Type
; left  : (∀ n, nth n env unit) → T
; right : (∀ n, nth n env unit) → T
; proof : ∀ g, left g = right g
}.

(** Example rewrite **)
Instance Eqn_hd T : OptEquation T :=
{ env     := T :: list T :: T :: nil
; left  g := hd (g 2) (cons (g 0) (g 1))
; right g := g 0
}. reflexivity.
Defined.
```

[6] Tate'10

Extended Optimization

- Recent work on extending optimizers[6]
- We already have the semantics & a language for reasoning

- Encode rewrites as records
- When can we rewrite?
  - Lambda? Cps? Clo?
  - ✓ More opportunities
  - ✗ More unpredictable
- Can we encode requirements?
  - Can we leverage Coq to prove them?

```
Class OptEquation T := OptRewrite
{ env   : list Type
; left  : (∀ n, nth n env unit) → T
; right : (∀ n, nth n env unit) → T
; proof : ∀ g, left g = right g
}.

(** Example rewrite **)
Instance Eqn_hd T : OptEquation T :=
{ env    := T :: list T :: T :: nil
; left g := hd (g 2) (cons (g 0) (g 1))
; right g := g 0
}. reflexivity.
Defined.
```

[6] Tate'10

Memory Representations

- Naïve memory representations are really bad
  - `Ascii` $=$ 9 words!
  - `nat` $= \sim 2n$ words!

Memory Representations

- Naïve memory representations are really bad
  - `Ascii` = 9 words!
  - `nat` = $\sim 2n$ words!

Never use `nat`?

Memory Representations

- Naïve memory representations are really bad
  - `Ascii` = 9 words!
  - `nat` = $\sim 2n$ words!
- This problem is already solved by extraction
  - `Extract Constant` & `Extract Inductive`
  - Not difficult to hook into these
  - What do we compile them to?

Memory Representations

- Naïve memory representations are really bad
  - `Ascii` = 9 words!
  - `nat` = $\sim 2n$ words!
- This problem is already solved by extraction
  - `Extract Constant` & `Extract Inductive`
  - Not difficult to hook into these
  - What do we compile them to?
- Can we do better?
  - Improvements seem to require real type information, including polymorphism.
  - Want to do it for copmiler data structures (like environments) too

Memory Representations

- Naïve memory representations are really bad
  - `Ascii` = 9 words!
  - `nat` = $\sim 2n$ words!
- This problem is already solved by extraction
  - `Extract Constant` & `Extract Inductive`
  - Not difficult to hook into these
  - What do we compile them to?
- Can we do better?
  - Improvements seem to require real type information, including polymorphism.
  - Want to do it for copmiler data structures (like environments) too
  - Hook into Ocaml extraction and get types?
    - Compilation of dependent types, e.g. cps translation, still an open problem!

Runtime

- LLVM based on C calling convention
  - Automatic management of the stack makes it difficult to control

`tail call f(k; 1, 2) ;`

Runtime

- LLVM based on C calling convention
  - Automatic management of the stack makes it difficult to control
  - Using a custom stack makes optimizations more difficult
    - GHC uses a custom alias analysis to improve optimization

`tail call f(k; 1, 2) ;`

Runtime

- LLVM based on C calling convention
  - Automatic management of the stack makes it difficult to control
  - Using a custom stack makes optimizations more difficult
    - GHC uses a custom alias analysis to improve optimization
- LLVM shadow stack is very heavy weight
  - Sub-optimal code-generation & minimal optimization
  - Compiler controlled stack would be helpful

`tail call f(k; 1, 2) ;`

What's Next?

Still lots to do.

- ExtLib
  - Are we using type classes effectively?
  - Will type classes scale?
  - What are the performance implications for Coq evaluation

Data Types with Type Classes

- We can do the same thing for abstract data types
- Several choices:

Container-style

```
Class CSet (T : Type) :=
{ set: Type
; has: T → set → bool
; add: T → set → set
}.
```

✓ Abstract type

✗ Can't write `map`

ExtLib-style

```
Class DSet (T : Type)
  (set : Type) :=
{ has: T → set → bool
; add: T → set → set
}.
```

✗ Transparent type

✓ Supports `map`

Haskell-style

```
Class PSet
  (set : Type → Type) :=
{ has: ∀ T,
  T → set T → bool
; add: ∀ T,
  T → set T → set T
}.
```

✓ Exposes functor

✗ Complex signature

✗ Need property on `T`

Can we layer the container-style on top of the ExtLib-style?

Reasoning about Data Types

- Separate, generic automation based on equational laws

- Separately track well-formedness
  - Avoids proofs in computation (0 computational overhead)
  - After-the-fact verification (`DSet_WF` only occurs in proofs)
  - Resolution picks the right proofs

```
Section dset.
  Variables T set : Type.
  Variable DS : DSet T set.

  Class DSetLaws : Type :=
  { DSet_WF : DS → Prop
  ; add_WF : ∀ s x,
    DSet_WF s → DSet_WF (add x s)
  ; has_add : ∀ s x,
    DSet_WF s → has x (add x s)
  ; ....
  }.
End dset.
```

Partial Functors

- Some functors are partial
  - Sets require equality or comparison
- Must expose the functorial nature to provide polymorphic operations

Total Functor

```
Class Monad (m : Type → Type) :=
{ bind : ∀ T U,
  m T → (T → m U) → m U
; ret : ∀ T, T → m T
}.
```

Default MonP

Partial Functor

```
Class PMonad (m : Type → Type) :=
{ MonP : Type → Type
; bind : ∀ T U, MonP T → MonP U →
  m T → (T → m U) → m U
; ret : ∀ T, MonP T → T → m T
}.
Existing Class MonP.

Class Any (T : Type) : Type := {}.
Global Instance Any_T T : Any T :=
    Build_Any T.
```

More Generic Reasoning

- Express more logical properties using type classes
  - Consistent name → generic automation
- Completely seemless to move between isomorphic types!

Examples

- RightIdent, LeftIdent
- Commutative, Associative
- Distributes
- ...others...

```
Ltac red_ident :=
  match goal with
    | ⊢ context [ ?F ?X _ ] ⇒
      let pf :=
      constr:(_ : LeftIdent F X)
      in rewrite pf
    | ...
  end.
```

How expensive is this?

Conclusions

Coq Compile

- Gallina $\rightarrow$ LLVM (via Scheme)
- Written in Gallina
- Native handling of I/O

ExtLib

- Library based on type classes
- Monads, abstract types
- After-the-fact reasoning

Compiler: `https://github.com/coq-ext-lib/coq-compile`
ExtLib: `https://github.com/coq-ext-lib/coq-ext-lib`

Contributors
ExtLib Gregory Malecha, Scott Moore, David Darais
Compiler Gregory Malecha, Scott Moore, Daniel Huang, Greg Morrisett, Lucas Waye, Gabby Ehrlich, Carl Jackson