Compositional and Customizable Reflective Proofs

Gregory Malecha
gmalecha@cs.harvard.edu

Harvard SEAS

October 2, 2013

Proof Engineering in Bedrock[1]

```
bfunction "lookup"("s", "k", "tmp") [lookupS]
  "s" ←∗ "s";;
   [∀ s, ∀ t,
    PRE[V] bst' s t (V "s") ∗ mallocHeap
    POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗
       mallocHeap]
  While ("s" ≠ 0) {
    "tmp" ← "s" + 4;;
    "tmp" ←∗ "tmp";;
    If ("k" = "tmp") {
      Return 1 (* Key matches! *)
    } else {
      If ("k" < "tmp") {
        (* Searching for a lower key *)
        "s" ←∗ "s"
      } else {
        (* Searching for a higher key *)
        "s" ← "s" + 8;;
        "s" ←∗ "s"
      }
    }
  };;
  Return 0
end

Theorem bstOk : moduleOk bst.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

- Composing automation
  1. Separation logic
  2. Symbolic execution
  3. Sets
  4. Bit-vectors

[1] Chlipala

Proof Engineering in Bedrock[1]

```
bfunction "lookup"("s", "k", "tmp") [lookupS]
  "s" ←∗ "s";;
   [∀ s, ∀ t,
    PRE[V] bst' s t (V "s") ∗ mallocHeap
    POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗
       mallocHeap]
  While ("s" ≠ 0) {
    "tmp" ← "s" + 4;;
    "tmp" ←∗ "tmp";;
    If ("k" = "tmp") {
      Return 1 (* Key matches! *)
```

```
vcgen; abstract (sep hints; auto).
```

```
        (* Searching for a lower key *)
        "s" ←∗ "s"
      } else {
        (* Searching for a higher key *)
        "s" ← "s" + 8;;
        "s" ←∗ "s"
      }
    }
  };;
  Return 0
end

Theorem bstOk : moduleOk bst.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

- Composing automation
  1. Separation logic
  2. Symbolic execution
  3. Sets
  4. Bit-vectors

[1] Chlipala

Proof Engineering in Bedrock[1]

```
bfunction "lookup"("s", "k", "tmp") [lookupS]
  "s" ←∗ "s";;
  [∀ s, ∀ t,
   PRE[V] bst' s t (V "s") ∗ mallocHeap
   POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗

    "tmp" ← "s" + 4;;
    "tmp" ←∗ "tmp";;
    If ("k" = "tmp") {
      Return 1 (* Key matches! *)

      (* Searching for a lower key *)
      "s" ←∗ "s"
    } else {
      (*
      "s"
      "s"
    }
  }
  };;
  Return 0
end

Theorem bstOk : moduleOk bst.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

Reflective automation

Customization

vcgen; abstract (sep hints; auto).

Integration with "manual" proofs

- Composing automation
  1. Separation logic
  2. Symbolic execution
  3. Sets
  4. Bit-vectors

[1] Chlipala

Proof Engineering in Bedrock[1]

```
bfunction "lookup"("s", "k", "tmp") [lookupS]
  "s" ←∗ "s";;
   [∀ s, ∀ t,
    PRE[V] bst' s t (V "s") ∗ mallocHeap
    POST[R] ⌈ (V "k" ∈ s) \is R ⌉ ∗ bst' s t (V "s") ∗
       mallocHeap]
  While ("s" ≠ 0) {
    "tmp" ← "s" + 4;;
    "tmp" ←∗ "tmp";;
    If ("k" = "tmp") {
      Return 1 (* Key matches! *)
    } else {
      If ("k" < "tmp") {
        (* Searching for a lower key *)
        "s" ←∗ "s"
      } else {
        (* Searching for a higher key *)
        "s" ← "s" + 8;;
        "s" ←∗ "s"
      }
    }
  };;
  Return 0
end

Theorem bstOk : moduleOk bst.
Proof. vcgen; abstract (sep hints; auto). Qed.
```

- Composing automation
  1. Separation logic
  2. Symbolic execution
  3. Sets
  4. Bit-vectors
- Also: data structures, thread scheduler, parser, web server, garbage collector, and more.
  - ≈15,000 LoC

[1] Chlipala

Outline

1. Reflective Proofs
2. Extensible Reflection
   - Universal Encoding
   - Binders
   - Unification
3. Future Work

Outline

1. Reflective Proofs

2. Extensible Reflection
   - Universal Encoding
   - Binders
   - Unification

3. Future Work

Proof by Reflection

$$\boxed{\langle \text{Goal} \rangle} \rightsquigarrow \boxed{\langle \text{Goal} \rangle}$$

$A * B * C = C * A * B$ $\qquad\qquad$ $1 = 1$

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$

`expr` `expr`

Syntax

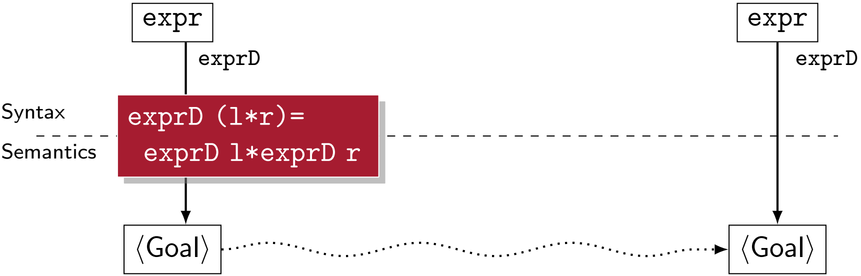
Semantics

$\boxed{\langle \text{Goal} \rangle} \rightsquigarrow \boxed{\langle \text{Goal} \rangle}$

$A * B * C = C * A * B$ $1 = 1$

Proof by Reflection

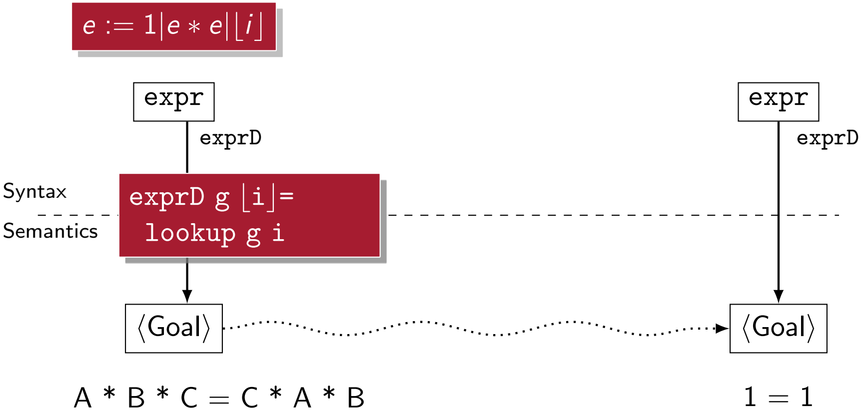
$e := 1 | e * e | \lfloor i \rfloor$



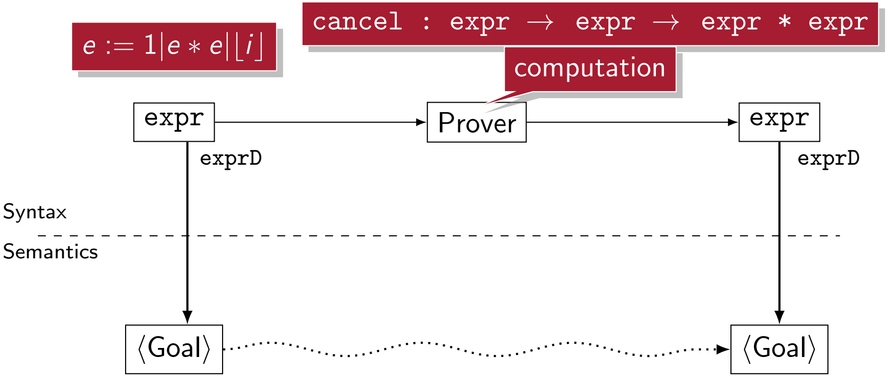
$A * B * C = C * A * B$

$1 = 1$

Proof by Reflection



Proof by Reflection



A * B * C = C * A * B

1 = 1

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$

computation

`expr` → Prover → `expr`

`exprD`

`exprD`

Syntax

Semantics

Prover Soundness

proof

commutative,
associative,
cancellative,
unit

$\langle \text{Goal} \rangle$

$\langle \text{Goal} \rangle$

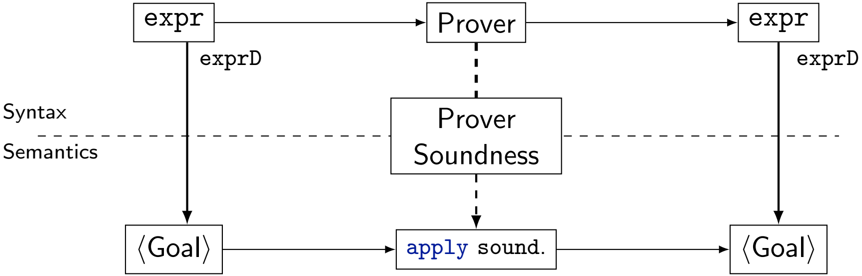
$$\begin{aligned}
&\mathit{sound} : \forall g\ l\ r\ l'\ r', \\
&\quad \text{cancel}\ l\ r = (l', r') \rightarrow \\
&\quad \text{exprD}\ g\ l' = \text{exprD}\ g\ r' \rightarrow \\
&\quad \text{exprD}\ g\ l = \text{exprD}\ g\ r
\end{aligned}$$

A * B * C = C *

$1 = 1$

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$

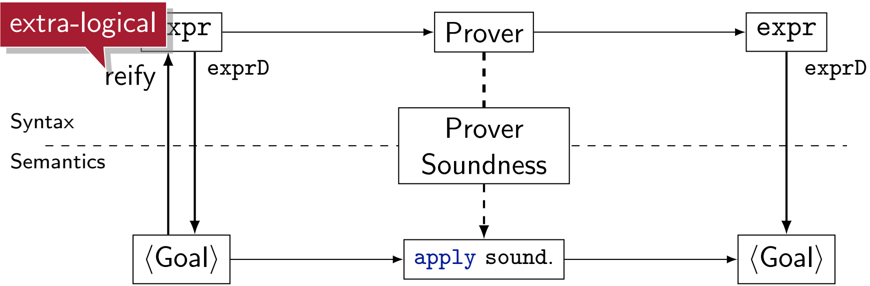


$A * B * C = C * A * B$

$1 = 1$

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$

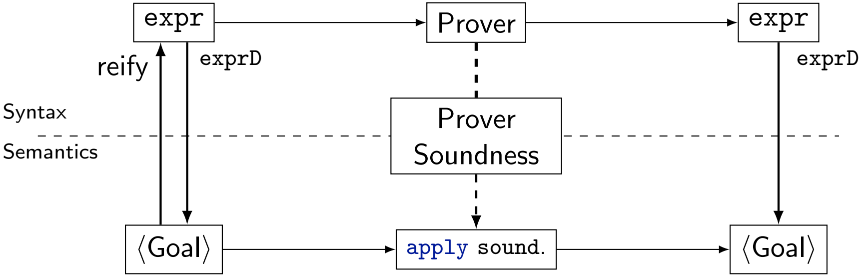


$A * B * C = C * A * B$

$1 = 1$

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$

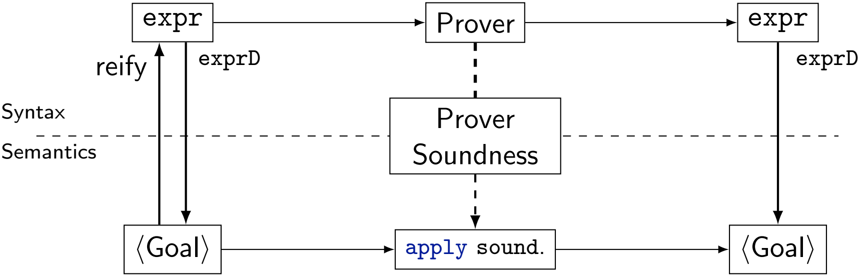


$P\ 2 * Q^{n+m} * R = R * P\ (1+1) * Q^n * Q^m$

$??? = ???$

Proof by Reflection

$e := 1 | e * e | \lfloor i \rfloor$



$P\, 2 * Q^{n+m} * R = R * P\, (1+1) * Q^n * Q^m$

$P\, 2 * Q^{n+m} = P\, (1+1) * Q^n * Q^m$

Outline

Environments for a Universal Encoding

Types

```
Inductive typ :=
| Arr (d r : typ)
| Typ (i : index).
```

Terms

```
Definition Func ts :=
{ t : typ & typD ts t }.
Inductive expr :=
| App (f x : expr) | ⌊ (i : index) ⌋
```

Environments for a Universal Encoding

Types

```
Inductive typ :=
| Arr (d r : typ)
| Typ (i : index).
```

Terms

```
Definition Func ts :=
{ t : typ & typD ts t }.
Inductive expr :=
| App (f x : expr) | ⌊ (i : index) ⌋
```

$1 = \lfloor i_{unit} \rfloor$
$x * y = \mathrm{App}\ (\mathrm{App}\ \lfloor i_{star} \rfloor\ x)\ y$

Environments for a Universal Encoding

Types

```
Inductive typ :=
| Arr (d r : typ) |Pro
| Typ (i : index).
```

Terms

```
Definition Func ts :=
{ t : typ & typD ts t }.
Inductive expr :=
| App (f x : expr) | ⌊(i : index)⌋
| Abs (t : typ) (e : expr)
| Var (i : nat)
| UVar (i : nat)
| Equal (t : typ)
| All (t : typ) | Ex (t : typ).
```

Reasoning with the Universal Encoding

```
Fixpoint simplify l r :=
  match l with
  | Star x y ⇒ ...
  | Unit ⇒ ...
  | ⌊_⌋ ⇒ ...
  end.
```

Reasoning with the Universal Encoding

```
Fixpoint simplify l r :=
  match l with
  | App (App⌊0⌋) x) y ⇒ …
  | ⌊1⌋ ⇒ …
  | _ ⇒ …
  end.
```

Environments

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

Reasoning with the Universal Encoding

```
Fixpoint simplify l r :=
  match l with
  | App (App⌊0⌋) x) y ⇒ ...
  | ⌊1⌋ ⇒ ...
  | _ ⇒ ...
  end.
```

```
Theorem simplify_sound : ∀ l r,
  simplify l r = (l',r') →
  exprD fs l' 0 =_M exprD fs r' 0 →
  exprD fs l 0 =_M exprD fs r 0.
Proof. ...  Qed.
```

exprD fs (App (App (Term 0) x) y) 0
≡ exprD fs x 0 * exprD fs y 0

Environments

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

- Fixing the environment enables reduction.

Reasoning with the Universal Encoding

```
Fixpoint simplify l r :=
  match l with
  | App (App⌊0⌋) x) y ⇒ ...
  | ⌊1⌋ ⇒ ...
  | _ ⇒ ...
  end.
```

```
Theorem simplify_sound : ∀ l r,
  simplify l r = (l',r') →
  exprD fs l' 0 =_M exprD fs r' 0 →
  exprD fs l 0 =_M exprD fs r 0.
Proof. ...  Qed.
```

exprD fs (App (App (Term 0) x) y) 0
≡ exprD fs x 0 * exprD fs y 0

Environments

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

- Fixing the environment enables reduction.

...but it prevents extension.

Retaining Extensibility

Complete Environment

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

Constrained Environments

```
Let tc := [ Some M ; Some nat ].
Let fc :=
  [ Some (0 ≫ 0 ≫ 0, Star_M)
  ; Some (0, Unit_M)
  ; Some (0 ≫ 1 ≫ 0, Exp_M) ].
```

Retaining Extensibility

Complete Environment

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0,  Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

"Applying" Constraints

```
Fixpoint repr (c : env (option T))
    (g : env T) : env T :=
  match c with
  | nil ⇒ g
  | Some t :: c ⇒ t :: repr c (tl g)
  | None :: c ⇒
    hd g :: repr c (tl g)
  end.
```

Constructors in head position

Constrained Environments

```
Let tc := [ Some M ; Some nat ].
Let fc  :=
  [ Some (0 ≫ 0 ≫ 0, Star_M)
  ; Some (0, Unit_M)
  ; Some (0 ≫ 1 ≫ 0, Exp_M) ].
```

Retaining Extensibility

Complete Environment

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

"Applying" Constraints

```
Fixpoint repr (c : env (option T))
    (g : env T) : env T :=
  match c with
  | nil ⇒ g
  | Some t :: c ⇒ t :: repr c (tl g)
  | None :: c ⇒
    hd g :: repr c (tl g)
  end.
```

Constrained Environments

M :: nat :: tl (tl ts)

```
Let tc := [ Some M ; Some nat ].
Let fc ts : env (Func (repr tc ts)) :=
  [ Some (0 ≫ 0 ≫ 0, Star_M)
  ; Some (0, Unit_M)
  ; Some (0 ≫ 1 ≫ 0, Exp_M) ].
```

Retaining Extensibility

Complete Environment

```
Let ts := [ M ; nat ].
Let fs := [ (0 ≫ 0 ≫ 0, Star_M)
          ; (0, Unit_M)
          ; (0 ≫ 1 ≫ 0, Exp_M) ].
```

```
Star_M :: Unit_M :: Exp_M
:: tl (tl (tl fs'))
```

“Applying Constraints”

```
Fixpoint repr (c : env (option T))
    (g : env T) : env T :=
  match c with
  | nil ⇒ g
  | Some t :: c ⇒ t :: repr c (tl g)
  | None :: c ⇒
    hd g :: repr c (tl g)
  end.
```

Constrained Environments

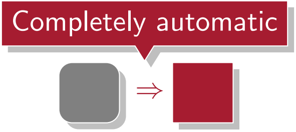
```
Let tc := [ Some M ; Some nat ].
Let fc ts : env (Func (repr tc ts)) :=
  [ Some (0 ≫ 0 ≫ 0, Star_M)
  ; Some (0, Unit_M)
  ; Some (0 ≫ 1 ≫ 0, Exp_M) ].

Theorem simplify_sound : ∀ ts' fs' l r,
  let fs := repr (fc ts') fs' in
  simplify l r = (l',r') →
  exprD fs l' 0 =_M exprD fs r' 0 →
  exprD fs l 0 =_M exprD fs r 0.
Proof. ... Qed.
```

Generic Reasoning: Binders & Lemmas

Lemma

`Lemma` **plus_le** : $\forall$ **n m**, **n** $\leq$ **n** $+$ **m**.
`Proof`. ... `Qed`.

Generic Reasoning: Binders & Lemmas



Lemma

```
Lemma plus_le : ∀ n m, n ≤ n + m.
Proof. ... Qed.
```

Reified Lemma

```
Definition Lem ts fs :=
  { e : expr & exprD fs e Pro }.

Let plus_le_lem ts fs
: Lem (repr .. ts) (repr .. fs) :=
 (All t_nat (All t_nat
    (Leq t_nat (Var 1)
         (Plus (Var 1) (Var 0)))),
  plus_le).
```

Generic Reasoning: Binders & Lemmas

Lemma

```
Lemma plus_le : ∀ n m, n ≤ n + m.
Proof. ... Qed.
```

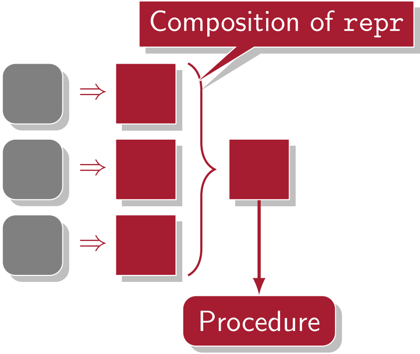
R

repr A (repr B g) ≡
repr B (repr A g)

```
De
  { e : expr & exprD fs e Pro }.
```

When A and B are compatible

```
Le
: Lem (repr .. ts) (repr .. fs) :=
 (All t_nat (All t_nat
    (Leq t_nat (Var 1)
         (Plus (Var 1) (Var 0)))),
  plus_le).
```



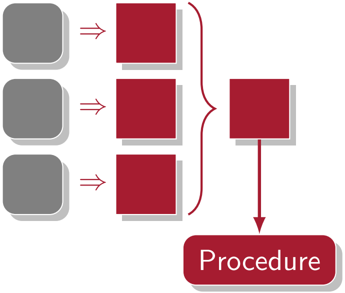
Generic Reasoning: Binders & Lemmas

Lemma

```
Lemma plus_le : ∀ n m, n ≤ n + m.
Proof. ... Qed.
```

Reified Lemma

```
Definition Lem ts fs :=
  { e : expr & exprD fs e Pro }.

Let plus_le_lem ts fs
: Lem (repr .. ts) (repr .. fs) :=
 (All t_nat (All t_nat
    (Leq t_nat (Var 1)
         (Plus (Var 1) (Var 0)))),
  plus_le).
```



- Reified "hint databases" usable by generic reasoning procedures
  - Reflective `auto`
  - (Setoid) `autorewrite`

Applying Lemmas & Supporting Unification Variables

$\forall n\, m, n \le n + m$

$$?\text{plus\_le}\ \frac{???}{x \le x + (y + z)}$$

Applying Lemmas & Supporting Unification Variables

$$?\text{plus\_le}\ \frac{???}{x \leq x + (y + z)}$$

$\forall n\, m, n \leq n + m$

"open" binders

$?1 \leq ?1 + ?2$

Applying Lemmas & Supporting Unification Variables

$$\text{plus\_le}\ \frac{}{x \leq x + (y + z)}$$

$\forall n\, m, n \leq n + m$

"open" binders

$?1 \leq ?1 + ?2$

Unification variables

unify the terms

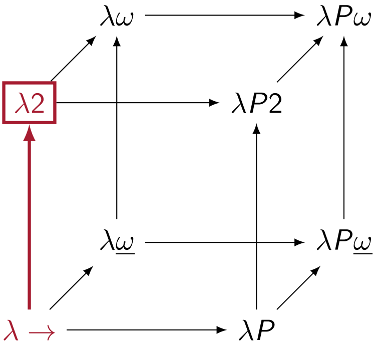
$\{?1 \mapsto x; ?2 \mapsto y + z\}$

Outline

Expressivity

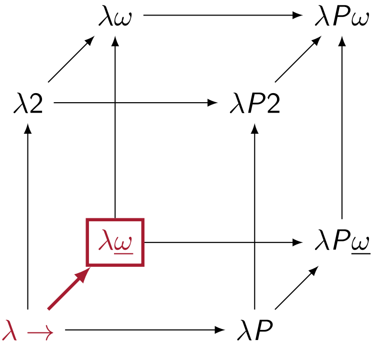
$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\boxed{\lambda \rightarrow} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Expressivity



- **Polymorphism**
  - Special constructors for $=$, $\forall$, $\exists$.
  - Otherwise, monomorphize.
  - Exploring enriching: $\lfloor - \rfloor_{t_1 \ldots t_n}$
    - ✗ Substitution and instantiation are not definitionally equal.

Expressivity



- **Type Functions**
  - Can support special cases (≫)
  - Currently monomorphize, e.g. `list`.

✗ General functor environment leads to universe inconsistencies.

  - Do we need universe polymorphism?

Expressivity

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda \to \arrow[uu] \arrow[ur] \arrow[rr] & & \boxed{\lambda P} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

- **Dependency**
  - ✗ Even fewer things will reduce.

  - Irrelevence of equality proofs?
  - Computational equality casts by isomorphisms?
  - Implicit type casts?

Feedback?

- **Framework Expressivity**
  - Polymorphism
  - Type functions
  - Dependent types
- **Applications**
  - Program verification (Bedrock[1], Charge![2†], Verifiable C[3†])
  - Monad reasoning[†]

Source Code

http://github.com/gmalecha/mirror-shard
http://github.com/gmalecha/mirror-core[†]

Thanks to collaborators: Adam Chlipala (MIT), Thomas Braibant (INRIA)

[1] Chlipala [2] Bengtson [3] Appel [†] In progress.

Composition & Parameterization

- Consistent constraints commute!

```
Let tc1 = [ Some M ; None ].
Let tc2 = [ Some M ; Some nat ].
```

$$
\begin{aligned}
\texttt{repr tc}_2\ (\texttt{repr tc}_1\ \texttt{ts}) &\equiv \texttt{repr tc}_2\ (\texttt{M} :: \texttt{hd}\ (\texttt{tl ts})\ \texttt{Empty\_set} :: \texttt{tl}\ (\texttt{tl ts})))\\
&\equiv \texttt{M} :: \texttt{nat} :: \texttt{tl}\ (\texttt{tl ts})\\
&\equiv \texttt{repr tc}_1\ (\texttt{M} :: \texttt{nat} :: \texttt{tl}\ (\texttt{tl ts})))\\
&\equiv \texttt{repr tc}_1\ (\texttt{repr tc}_2\ \texttt{ts})
\end{aligned}
$$